(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 4 109 247 A1

(12) EUROPEAN PATENT APPLICATION

(43) Date of publication:
28.12.2022 Bulletin 2022/52

(21) Application number: **22181066.6**

(22) Date of filing: **24.06.2022**

(51) International Patent Classification (IPC):
G06F 9/30 *(2018.01)*

(52) Cooperative Patent Classification (CPC):
G06F 9/30036; G06F 9/3001; G06F 9/30025

(84) Designated Contracting States:
AL AT BE BG CH CY CZ DE DK EE ES FI FR GB GR HR HU IE IS IT LI LT LU LV MC MK MT NL NO PL PT RO RS SE SI SK SM TR
Designated Extension States:
BA ME
Designated Validation States:
KH MA MD TN

(30) Priority: **26.06.2021 US 202117359519**

(71) Applicant: **INTEL Corporation
Santa Clara, CA 95054 (US)**

(72) Inventors:
- **ADELMAN, Menachem
7179034 Modi'in (IL)**
- **VALENTINE, Robert
36054 Kiryat Tivon (IL)**
- **SPERBER, Zeev
3092832 Zichron Yackov (IL)**
- **GRADSTEIN, Amit
3052316 Binyamina (IL)**
- **RUBANOVICH, Simon
34792 Haifa (IL)**
- **MELLER, Sagi
3091885 Zichron Yaakov (IL)**
- **HUGHES, Christopher
Santa Clara, CA, 95051 (US)**
- **GEORGANAS, Evangelos
San Jose, CA, 95129 (US)**
- **HEINECKE, Alexander
San Jose, CA, 95134 (US)**
- **CHARNEY, Mark
Lexington, MA, 02421 (US)**

(74) Representative: **Samson & Partner Patentanwälte mbB
Widenmayerstraße 6
80538 München (DE)**

(54) APPARATUSES, METHODS, AND SYSTEMS FOR INSTRUCTIONS FOR MATRIX MULTIPLICATION INSTRUCTIONS

(57) Techniques for matrix multiplication are described. In some examples, decode circuitry is to decode a single instruction having fields for an opcode, an indication of a location of a first source operand, an indication of a location of a second source operand, and an indication of a location of a destination operand, wherein the opcode is to indicate that execution circuitry is to at least convert data elements of the first and second source operands from a first floating point representation to a second floating point representation, perform matrix multiplication with the converted data elements, and accumulate results of the matrix multiplication in the destination operand in the first floating point representation; and the execution circuitry is to execute to the decoded instruction as specified by the opcode.

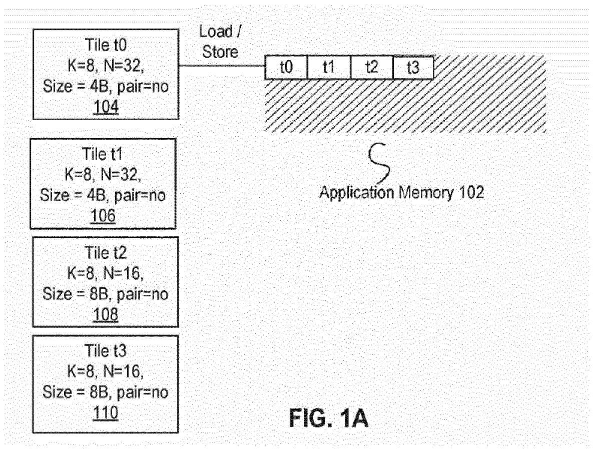


FIG. 1A

EP 4 109 247 A1

Description

TECHNICAL FIELD

[0001] The disclosure relates generally to computer processor architecture, and, more specifically, to systems and methods for performing matrix multiplication instructions.

BACKGROUND

[0002] Matrices are increasingly important in many computing tasks such as machine learning and other bulk data processing. Deep Learning is a class of machine learning algorithms. Deep learning architectures, such as deep neural networks, have been applied to fields including computer vision, speech recognition, natural language processing, audio recognition, social network filtering, machine translation, bioinformatics and drug design.

[0003] Inference and training, two tools used for deep learning, are tending towards low precision arithmetic. Maximizing throughput of deep learning algorithms and computations may assist in meeting the needs of deep learning processors, for example, those performing deep learning in a data center.

[0004] Matrix-matrix multiplication (a.k.a., GEMM or General Matrix Multiplication) is a common compute-heavy operation on modern processors. Special hardware for matrix multiplication (e.g., GEMM) is a good option for improving the peak compute (and energy efficiency) of certain applications, such as deep learning.

[0005] Some of these applications, including deep learning, can operate on input data elements with relatively few bits without losing accuracy, as long as the output elements have enough bits (i.e., more than the inputs).

BRIEF DESCRIPTION OF THE DRAWINGS

[0006] The present disclosure is illustrated by way of example and not limitation in the figures of the accompanying drawings, in which like references indicate similar elements and in which:

Figure 1A illustrates an embodiment of configured tiles.
Figure 1B illustrates an embodiment of configured tiles.
Figure 2 illustrates several examples of matrix storage.
Figure 3 illustrates an embodiment of a system utilizing a matrix (tile) operations accelerator.
Figures 4 and 5 show different embodiments of how memory is shared using a matrix operations accelerator.
Figure 6 illustrates an embodiment of matrix multiply accumulate operation using tiles ("TMMA").
Figure 7 illustrates an embodiment of a subset of the execution of an iteration of a chained fused multiply accumulate instruction.
Figure 8 illustrates an embodiment of a subset of the execution of an iteration of a chained fused multiply accumulate instruction.
Figure 9 illustrates an embodiment of a subset of the execution of an iteration of a chained fused multiply accumulate instruction.
Figure 10 illustrates an embodiment of a subset of the execution of an iteration of chained fused multiply accumulate instruction.
Figure 11 illustrates power-of-two sized SIMD implementations wherein the accumulators use input sizes that are larger than the inputs to the multipliers according to an embodiment.
Figure 12 illustrates an embodiment of a system utilizing matrix operations circuitry.
Figure 13 illustrates an embodiment of a processor core pipeline supporting matrix operations using tiles.
Figure 14 illustrates an embodiment of a processor core pipeline supporting matrix operations using tiles.
Figure 15 illustrates an example of a matrix expressed in row major format and column major format.
Figure 16 illustrates an example of usage of matrices (tiles).
Figure 17 illustrates an embodiment a method of usage of matrices (tiles).
Figure 18 illustrates support for configuration of the usage of tiles according to an embodiment.
Figure 19 illustrates an embodiment of a description of the matrices (tiles) to be supported.
Figures 20(A)-(D) illustrate examples of register(s).
Figure 21 illustrates an exemplary execution of a tile matrix multiplication on FP19 data elements instruction according to some embodiments.
Figure 22 illustrates an embodiment of method to process a TMMULFP19PS instruction.
Figure 23 illustrates embodiments of execution of a TMMULFP19PS instruction.
Figure 24 illustrates embodiments pseudo-code for the execution of TMMULFP19PS instruction.
Figure 25 illustrates an exemplary execution of a tile matrix transpose and multiplication on FP19 data elements

instruction according to some embodiments.

Figure 26 illustrates an embodiment of method to process a TTMULFP19PS instruction.

Figure 27 illustrates embodiments of execution of a TTMULFP19PS instruction.

Figure 28 illustrates embodiments of pseudo-code for the execution of a TTMULFP19PS instruction.

Figure 29 illustrates embodiments of hardware to process an instruction such as a TTMULFP19PS and/or TMMULFP19PS instruction.

Figure 30 illustrates embodiments of an exemplary system.

Figure 31 illustrates a block diagram of embodiments of a processor that may have more than one core, may have an integrated memory controller, and may have integrated graphics.

Figure 32(A) is a block diagram illustrating both an exemplary in-order pipeline and an exemplary register renaming, out-of-order issue/execution pipeline according to embodiments of the invention.

Figure 32(B) is a block diagram illustrating both an exemplary embodiment of an in-order architecture core and an exemplary register renaming, out-of-order issue/execution architecture core to be included in a processor according to embodiments of the invention.

Figure 33 illustrates embodiments of execution unit(s) circuitry, such as execution unit(s) circuitry of Figure 32(B).

Figure 34 is a block diagram of a register architecture according to some embodiments.

Figure 35 illustrates embodiments of an instruction format.

Figure 36 illustrates embodiments of an addressing field.

Figure 37 illustrates embodiments of a first prefix.

FIGS. 38(A)-(D) illustrate embodiments of how the R, X, and B fields of the first prefix 3501(A) are used.

Figures 39(A)-(B) illustrate embodiments of a second prefix.

Figure 40 illustrates embodiments of a third prefix.

Figure 41 illustrates a block diagram contrasting the use of a software instruction converter to convert binary instructions in a source instruction set to binary instructions in a target instruction set according to embodiments of the invention.

DETAILED DESCRIPTION

[0007] In the following description, numerous specific details are set forth. However, it is understood that embodiments may be practiced without these specific details. In other instances, well-known circuits, structures and techniques have not been shown in detail in order not to obscure the understanding of this description.

[0008] References in the specification to "one embodiment," "an embodiment," "an example embodiment," etc., indicate that the embodiment described may include a particular feature, structure, or characteristic, but every embodiment may not necessarily include the particular feature, structure, or characteristic. Moreover, such phrases are not necessarily referring to the same embodiment. Further, when a particular feature, structure, or characteristic is described in connection with an embodiment, it is submitted that it is within the knowledge of one skilled in the art to affect such feature, structure, or characteristic in connection with other embodiments whether or not explicitly described.

[0009] In many mainstream processors, handling matrices is a difficult and/or instruction intensive task. For example, rows of a matrix could be put into a plurality of packed data (e.g., SIMD or vector) registers and then operated on individually. For example, an add two 8x2 matrices may require a load or gather into four packed data registers depending upon data sizes. Then a first add of packed data registers corresponding to a first row from each matrix is performed and a second add of packed data registers corresponding to a second row from each matrix is performed. Then the resulting packed data registers are scattered back to memory. While for small matrices this scenario may be acceptable, it is often not acceptable with larger matrices.

DISCUSSION

[0010] Described herein are mechanisms to support matrix operations in computer hardware such as central processing units (CPUs), graphic processing units (GPUs), and accelerators. The matrix operations utilize 2-dimensional (2-D) data structures representing one or more packed regions of memory such as registers. Throughout this description, these 2-D data structures are referred to as tiles. Note that a matrix may be smaller than a tile (use less than all of a tile) or utilize a plurality of tiles (the matrix is larger than the size of any one tile). Throughout the description, matrix (tile) language is used to indicate operations performed using tiles that impact a matrix; whether or not that matrix is larger than any one tile is not typically relevant.

[0011] Each tile may be acted upon by different operations such as those that are detailed herein and include, but are not limited to: matrix (tile) multiplication, tile add, tile subtract, tile diagonal, tile zero, tile transform, tile dot product, tile broadcast, tile row broadcast, tile column broadcast, tile multiplication, tile multiplication and accumulation, tile move, etc. Additionally, support for operators such as the use of a scale and/or bias may be used with these operations or in support of non-numeric applications in the future, for instance, OpenCL "local memory," data compression/decompression, etc. Also described herein are instructions for performing matrix (tile) 16-bit tile dot product (TILEDPFP16PS) instructions.

[0012] Portions of storage (such as memory (non-volatile and volatile), registers, cache, etc.) are arranged into tiles of different horizontal and vertical dimensions. For example, a tile may have horizontal dimension of 4 (e.g., four rows of a matrix) and a vertical dimension of 8 (e.g., 8 columns of the matrix). Typically, the horizontal dimension is related to element sizes (e.g., 2-, 4-, 8-, 16-, 32-, 64-, 128-bit, etc.). Multiple datatypes (single precision floating-point, double precision floating-point, integer, etc.) may be supported.

EXEMPLARY USAGE OF CONFIGURED TILES

[0013] In some embodiments, tile parameters can be configured. For example, a given tile may be configured to provide tile options. Exemplary tile options include but are not limited to: a number of rows of the tile, a number of columns of the tile, whether the tile is VALID, and/or whether the tile consists of a PAIR of equal-sized tiles.

[0014] **Figure 1A** illustrates an embodiment of configured tiles. As shown, 4 kB of application memory 102 have stored thereon 4 1kB titles, tile t0 104, tile t1 106, tile t2 108, and tile t3 110. In this example, the 4 tiles do not consist of pairs, and each have elements arranged in rows and columns. Tile t0 104 and tile t1 106 have K rows and N columns of 4-byte elements (e.g., single precision data), where K equals 8 and N=32. Tile t2 108 and tile t3 110 have K rows and N/2 columns of 8-byte elements (e.g., double precision data). As the double precision operands are twice the width of single precision, this configuration is consistent with a palette, used to provide tile options, supplying at least 4 names with total storage of at least 4 kB. In operation, the tiles can be loaded from and stored to memory using load and store operations. Depending upon the instruction encoding scheme used, the amount of available application memory, as well as the size, number, and configuration of available tiles varies.

[0015] **Figure 1B** illustrates an embodiment of configured tiles. As shown, 4 kB of application memory 122 have stored thereon 2 pairs of 1kB-titles, the first pair being tile t4L 124 and tile t4R 126, and the second pair being tile t5L 128 and tile t5R 130. As shown the pairs of tiles are divided into a left tile and a right tile. In other embodiments, the pair of tiles are divided into an even tile and an odd tile. In this example, the 4 tiles each have elements arranged in rows and columns. Tile t4L 124 and tile t4R 126 have K rows and N columns of 4-byte elements (e.g., single precision floating-point data), where K equals 8 and N equals 32. Tile t5L 128 and tile t5R 130 have K rows and N/2 columns of 8-byte elements (e.g., double precision floating-point data). As the double precision operands are twice the width of single precision, this configuration is consistent with a palette, used to provide tile options, supplying at least 2 names with total storage of at least 4 kB. The four tiles of **Figure 1A** use 4 names, each naming a 1 kB tile, whereas the 2 pairs of tiles in **Figure 1B** can use 2 names to specify the paired tiles. In some embodiments, tile instructions accept a name of a paired tile as an operand. In operation, the tiles can be loaded from and stored to memory using load and store operations. Depending upon the instruction encoding scheme used, the amount of available application memory, as well as the size, number, and configuration of available tiles varies.

[0016] In some embodiments, tile parameters are definable. For example, a "palette" is used to provide tile options. Exemplary options include, but are not limited to: the number of tile names, the number of bytes in a row of storage, the number of rows and columns in a tile, etc. For example, a maximum "height" (number of rows) of a tile may be defined as:

[0017] Tile Max Rows = Architected Storage / (The Number of Palette Names * The Number of Bytes per row).

[0018] As such, an application can be written such that a fixed usage of names will be able to take advantage of different storage sizes across implementations.

[0019] Configuration of tiles is done using a matrix (tile) configuration ("TILECONFIG") instruction, where a particular tile usage is defined in a selected palette. This declaration includes the number of tile names to be used, the requested number of rows and columns per name (tile), and, in some embodiments, the requested datatype of each tile. In some embodiments, consistency checks are performed during the execution of a TILECONFIG instruction to determine that it matches the restrictions of the palette entry.

EXEMPLARY TILE STORAGE TYPES

[0020] **Figure 2** illustrates several examples of matrix storage. In (A), a tile is stored in memory. As shown, each "row" consists of four packed data elements. To get to the next "row," a stride value is used. Note that rows may be consecutively stored in memory. Strided memory accesses allow for access of one row to then next when the tile storage does not map the underlying memory array row width.

[0021] Tile loads from memory and stores to memory are typically strided accesses from the application memory to packed rows of data. Exemplary TILELOAD and TILESTORE instructions, or other instruction references to application memory as a TILE operand in load-op instructions, are, in some embodiments, restartable to handle (up to) 2*rows of page faults, unmasked floating-point exceptions, and/or interrupts per instruction.

[0022] In (B), a matrix is stored in a tile comprised of a plurality of registers such as packed data registers (single instruction, multiple data (SIMD) or vector registers). In this example, the tile is overlaid on three physical registers. Typically, consecutive registers are used, however, this need not be the case.

[0023] In (C), a matrix is stored in a tile in non-register storage accessible to a fused multiply accumulate (FMA) circuit used in tile operations. This storage may be inside of an FMA, or adjacent to it. Additionally, in some embodiments, discussed below, the storage may be for a data element and not an entire row or tile.

[0024] The supported parameters for the TMMA architecture are reported via CPUID. In some embodiments, the list of information includes a maximum height and a maximum SIMD dimension. Configuring the TMMA architecture requires specifying the dimensions for each tile, the element size for each tile and the palette identifier. This configuration is done by executing the TILECONFIG instruction.

[0025] Successful execution of a TILECONFIG instruction enables subsequent TILE operators. A TILERELEASEALL instruction clears the tile configuration and disables the TILE operations (until the next TILECONFIG instructions executes). In some embodiments, XSAVE, XSTORE, etc. are used in context switching using tiles. In some embodiments, 2 XCR0 bits are used in XSAVE, one for TILECONFIG metadata and one bit corresponding to actual tile payload data.

[0026] TILECONFIG not only configures the tile usage, but also sets a state variable indicating that the program is in a region of code with tiles configured. An implementation may enumerate restrictions on other instructions that can be used with a tile region such as no usage of an existing register set, etc.

[0027] Exiting a tile region is typically done with the TILERELEASEALL instruction. It takes no parameters and swiftly invalidates all tiles (indicating that the data no longer needs any saving or restoring) and clears the internal state corresponding to being in a tile region.

[0028] In some embodiments, tile operations will zero any rows and any columns beyond the dimensions specified by the tile configuration. For example, tile operations will zero the data beyond the configured number of columns (factoring in the size of the elements) as each row is written. For example, with 64-byte rows and a tile configured with 10 rows and 12 columns, an operation writing FP32 elements would write each of the first 10 rows with 12*4 bytes with output/result data and zero the remaining 4*4 bytes in each row. Tile operations also fully zero any rows after the first 10 configured rows. When using 1K tile with 64-byte rows, there would be 16 rows, so in this example, the last 6 rows would also be zeroed.

[0029] In some embodiments, a context restore instruction (e.g., XRSTOR), when loading data, enforces that the data beyond the configured rows for a tile will be maintained as zero. If there is no valid configuration, all rows are zeroed. XRSTOR of tile data can load garbage in the columns beyond those configured. It should not be possible for XRSTOR to clear beyond the number of columns configured because there is not an element width associated with the tile configuration.

[0030] Context save (e.g., XSAVE) exposes the entire TILE storage area when writing it to memory. If XRSTOR loaded garbage data into the rightmost part of a tile, that data will be saved by XSAVE. XSAVE will write zeros for rows beyond the number specified for each tile.

[0031] In some embodiments, tile instructions are restartable. The operations that access memory allow restart after page faults. The computational instructions that deal with floating-point operations also allow for unmasked floating-point exceptions, with the masking of the exceptions controlled by a control and/or status register.

[0032] To support restarting instructions after these events, the instructions store information in the start registers detailed below.

MATRIX (TILE) OPERATION SYSTEMS

EXEMPLARY HARDWARE SUPPORT

[0033] **Figure 3** illustrates an embodiment of a system utilizing a matrix (tile) operations accelerator. In this illustration, a host processor/processing system 301 communicates commands 311 (e.g., matrix manipulation operations such as arithmetic or matrix manipulation operations, or load and store operations) to a matrix operations accelerator 307. However, this is shown this way for discussion purposes only. As detailed later, this accelerator 307 may be a part of a processing core. Typically, commands 311 that are tile manipulation operator instructions will refer to tiles as register-register ("reg-reg") or register-memory ("reg-mem") format. Other commands such as TILESTORE, TILELOAD, TILECONFIG, etc., do not perform data operations on a tile. Commands may be decoded instructions (e.g., micro-ops) or macroinstructions for the accelerator 307 to handle.

[0034] In this example, a coherent memory interface 303 is coupled to the host processor/processing system 301 and matrix operations accelerator 307 such that they can share memory. **Figures 4 and 5** show different embodiments of how memory is shared using a matrix operations accelerator. As shown in **Figure 4,** the host processor 401 and matrix operations accelerator circuitry 405 share the same memory 403. **Figure 5** illustrates an embodiment where the host processor 501 and matrix operations accelerator 505 do not share memory but can access each other's memory. For example, processor 501 can access tile memory 507 and utilize its host memory 503 as normal. Similarly, the matrix operations accelerator 505 can access host memory 503, but more typically uses its own memory 507. Note these memories may be of different types.

[0035] In some embodiments, tiles are supported using an overlay over physical registers. For example, a tile may utilize 16 1,024-bit registers, 32 512-bit registers, etc. depending on the implementation. In some embodiments, the matrix operations utilize 2-dimensional (2-D) data structures representing one or more packed regions of memory such as registers. Throughout this description, these 2-D data structures are referred to as tiles or tile registers.

[0036] In some embodiments, the matrix operations accelerator 307 includes a plurality of FMAs 309 coupled to data buffers 305 (in some implementations, one or more of these buffers 305 are stored in the FMAs of the grid as shown). The data buffers 305 buffer tiles loaded from memory and/or tiles to be stored to memory (e.g., using a tileload or tilestore instruction). Data buffers may be, for example, a plurality of registers. Typically, these FMAs are arranged as a grid of chained FMAs 309 which are able to read and write tiles. In this example, the matrix operations accelerator 307 is to perform a matrix multiply operation using tiles T0, T1, and T2. At least one of tiles is housed in the FMA grid 309. In some embodiments, all tiles in an operation are stored in the FMA grid 309. In other embodiments, only a subset is stored in the FMA grid 309. As shown, T1 is housed and T0 and T2 are not. Note that A, B, and C refer to the matrices of these tiles which may or may not take up the entire space of the tile.

[0037] **Figure 6** illustrates an embodiment of matrix multiply accumulate operation using tiles ("TMMA").

[0038] The number of rows in the matrix (TILE A 601) matches the number of serial (chained) FMAs comprising the computation's latency. An implementation is free to recirculate on a grid of smaller height, but the computation remains the same.

[0039] The source/destination vector comes from a tile of N rows (TILE C 605) and the grid of FMAs 611 performs N vector-matrix operations resulting in a complete instruction performing a matrix multiplication of tiles. Tile B 603 is the other vector source and supplies "broadcast" terms to the FMAs in each stage.

[0040] In operation, in some embodiments, the elements of matrix B (stored in a tile B 603) are spread across the rectangular grid of FMAs. Matrix B (stored in tile A 601) has its elements of a row transformed to match up with the columnar dimension of the rectangular grid of FMAs. At each FMA in the grid, an element of A and B are multiplied and added to the incoming summand (from above in the Figure) and the outgoing sum is passed to the next row of FMAs (or the final output).

[0041] The latency of a single step is proportional to K (row height of matrix B) and dependent TMMAs typically have enough source-destination rows (either in a single tile or across tile) to hide that latency. An implementation may also split the SIMD (packed data element) dimension M (row height of matrix A) across time steps, but this simply changes the constant that K is multiplied by. When a program specifies a smaller K than the maximum enumerated by the TMACC, an implementation is free to implement this with "masking" or "early outs."

[0042] The latency of an entire TMMA is proportional to N*K. The repeat rate is proportional to N. The number of MACs per TMMA instruction is N*K*M.

[0043] **Figure 7** illustrates an embodiment of a subset of the execution of an iteration of a chained fused multiply accumulate instruction. In particular, this illustrates execution circuitry of an iteration of one packed data element position of the destination. In this embodiment, the chained fused multiply accumulate is operating on signed sources wherein the accumulator is 2× the input data size.

[0044] A first signed source (source 1 701) and a second signed source (source 2 703) each have four packed data elements. Each of these packed data elements stores signed data such as floating-point data. A third signed source (source 3 709) has two packed data elements, each of which stores signed data. The sizes of the first and second signed sources 701 and 703 are half that of the third signed source (initial value or previous result) 709. For example, the first and second signed sources 701 and 703 could have 32-bit packed data elements (e.g., single precision floating-point) while the third signed source 709 could have 64-bit packed data elements (e.g., double precision floating-point).

[0045] In this illustration, only the two most significant packed data element positions of the first and second signed sources 701 and 703 and the most significant packed data element position of the third signed source 709 are shown. Of course, the other packed data element positions would also be processed.

[0046] As illustrated, packed data elements are processed in pairs. For example, the data of the most significant packed data element positions of the first and second signed sources 701 and 703 are multiplied using a multiplier circuit 705, and the data from second most significant packed data element positions of the first and second signed sources 701 and 703 are multiplied using a multiplier circuit 707. In some embodiments, these multiplier circuits 705 and 707 are reused for other packed data elements positions. In other embodiments, additional multiplier circuits are used so that the packed data elements are processed in parallel. In some contexts, parallel execution is done using lanes that are the size of the signed third source 709. The results of each of the multiplications are added using addition circuitry 711.

[0047] The result of the addition of the results of the multiplications is added to the data from most significant packed data element position of the signed source 3 709 (using a different adder 713 or the same adder 711).

[0048] Finally, the result of the second addition is either stored into the signed destination 715 in a packed data element position that corresponds to the packed data element position used from the signed third source 709 or passed on to the next iteration if there is one. In some embodiments, a writemask is applied to this storage such that if a corresponding writemask (bit) is set, the storage happens, and, if not set, the storage does not happen.

[0049] **Figure 8** illustrates an embodiment of a subset of the execution of an iteration of a chained fused multiply accumulate instruction. In particular, this illustrates execution circuitry of an iteration of one packed data element position of the destination. In this embodiment, the chained fused multiply accumulate is operating on signed sources wherein the accumulator is 2x the input data size.

[0050] A first signed source (source 1 801) and a second signed source (source 2 803) each have four packed data elements. Each of these packed data elements stores signed data such as integer data. A third signed source (source 3 809) has two packed data elements, each of which stores signed data. The sizes of the first and second signed sources 801 and 803 are half that of the third signed source 809. For example, the first and second signed sources 801 and 803 could have 32-bit packed data elements (e.g., single precision floating-point) the third signed source 809 could have 64-bit packed data elements (e.g., double precision floating-point).

[0051] In this illustration, only the two most significant packed data element positions of the first and second signed sources 801 and 803 and the most significant packed data element position of the third signed source 809 are shown. Of course, the other packed data element positions would also be processed.

[0052] As illustrated, packed data elements are processed in pairs. For example, the data of the most significant packed data element positions of the first and second signed sources 801 and 803 are multiplied using a multiplier circuit 805, and the data from second most significant packed data element positions of the first and second signed sources 801 and 803 are multiplied using a multiplier circuit 807.In some embodiments, multiplier circuits 805 and 807 perform the multiplications with infinite precision without saturation and use adder/saturation circuitry 813 to saturate the results of the accumulation to plus or minus infinity in case of an overflow and to zero in case of any underflow. In other embodiments, multiplier circuits 805 and 807 perform the saturation themselves. In some embodiments, these multiplier circuits 805 and 807 are reused for other packed data element positions. In other embodiments, additional multiplier circuits are used so that the packed data elements are processed in parallel. In some contexts, parallel execution is done using lanes that are the size of the signed third source (initial value or previous iteration result) 809. The results of each of the multiplications are added to the signed third source 809 using addition/saturation circuitry 813.

[0053] Addition/saturation (accumulator) circuitry 813 preserves a sign of an operand when the addition results in a value that is too big. In particular, saturation evaluation occurs on the infinite precision result between the multi-way-add and the write to the destination or next iteration. When the accumulator 813 is floating-point and the input terms are integer, the sum of products and the floating-point accumulator input value are turned into infinite precision values (fixed point numbers of hundreds of bits), the addition of the multiplication results and the third input is performed, and a single rounding to the actual accumulator type is performed.

[0054] Unsigned saturation means the output values are limited to a maximum unsigned number for that element width (all Is). Signed saturation means a value is limited to the be in the range between a minimum negative number and a max positive number for that element width (for bytes for example, the range is from -128 (= - 2^7) to 127(=2^7-1)).

[0055] The result of the addition and saturation check is stored into the signed result 815 in a packed data element position that corresponds to the packed data element position used from the signed third source 809 or passed on to the next iteration if there is one. In some embodiments, a writemask is applied to this storage such that if a corresponding writemask (bit) is set, the storage happens, and, if not set, the storage does not happen.

[0056] **Figure 9** illustrates an embodiment of a subset of the execution of an iteration of a chained fused multiply accumulate instruction. In particular, this illustrates execution circuitry of an iteration of one packed data element position of the destination. In this embodiment, the chained fused multiply accumulate is operating on a signed source and an unsigned source wherein the accumulator is 4x the input data size.

[0057] A first signed source (source 1 901) and a second unsigned source (source 2 903) each have four packed data elements. Each of these packed data elements has data such as floating-point or integer data. A third signed source (initial value or result 915) has a packed data element of which stores signed data. The sizes of the first and second sources 901 and 903 are a quarter of the third signed source 915. For example, the first and second sources 901 and 903 could have 16-bit packed data elements (e.g., word) and the third signed source 915 could have 64-bit packed data elements (e.g., double precision floating-point or 64-bit integer).

[0058] In this illustration, the four most significant packed data element positions of the first and second sources 901 and 903 and the most significant packed data element position of the third signed source 915 are shown. Of course, other packed data element positions would also be processed if there are any.

[0059] As illustrated, packed data elements are processed in quadruplets. For example, the data of the most significant packed data element positions of the first and second sources 901 and 903 are multiplied using a multiplier circuit 905, data from second most significant packed data element positions of the first and second sources 901 and 903 are multiplied using a multiplier circuit 907, data from third most significant packed data element positions of the first and second sources 901 and 903 are multiplied using a multiplier circuit 909, and data from the least significant packed data element positions of the first and second sources 901 and 903 are multiplied using a multiplier circuit 911. In some embodiments, the signed packed data elements of the first source 901 are sign extended and the unsigned packed data elements of the second source 903 are zero extended prior to the multiplications.

[0060] In some embodiments, these multiplier circuits 905-911 are reused for other packed data elements positions. In other embodiments, additional multiplier circuits are used so that the packed data elements are processed in parallel. In some contexts, parallel execution is done using lanes that are the size of the signed third source 915. The results of each of the multiplications are added using addition circuitry 913.

[0061] The result of the addition of the results of the multiplications is added to the data from most significant packed data element position of the signed source 3 915 (using a different adder 917 or the same adder 913).

[0062] Finally, the result 919 of the second addition is either stored into the signed destination in a packed data element position that corresponds to the packed data element position used from the signed third source 915 or passed to the next iteration. In some embodiments, a writemask is applied to this storage such that if a corresponding writemask (bit) is set, the storage happens, and, if not set, the storage does not happen.

[0063] **Figure 10** illustrates an embodiment of a subset of the execution of an iteration of chained fused multiply accumulate instruction. In particular, this illustrates execution circuitry of an iteration of one packed data element position of the destination. In this embodiment, the chained fused multiply accumulate is operating on a signed source and an unsigned source wherein the accumulator is 4x the input data size.

[0064] A first signed source 1001 and a second unsigned source 1003 each have four packed data elements. Each of these packed data elements stores data such as floating-point or integer data. A third signed source 1015 (initial or previous result) has a packed data element of which stores signed data. The sizes of the first and second sources are a quarter of the third signed source 1015 (initial or previous result). For example, the first and second sources could have 16-bit packed data elements (e.g., word) and the third signed source 1015 (initial or previous result) could have 64-bit packed data elements (e.g., double precision floating-point or 64-bit integer).

[0065] In this illustration, the four most significant packed data element positions of the first signed source 1001 and the second unsigned source 1003 and the most significant packed data element position of the third signed source 1015 are shown. Of course, other packed data element positions would also be processed if there are any.

[0066] As illustrated, packed data elements are processed in quadruplets. For example, the data of the most significant packed data element positions of the first signed source 1001 and the second unsigned source 1003 are multiplied using a multiplier circuit 1005, data from second most significant packed data element positions of the first signed source 1001 and the second unsigned source 1003 are multiplied using a multiplier circuit 1007, data from third most significant packed data element positions of the first signed source 1001 and the second unsigned source 1003 are multiplied using a multiplier circuit 1009, and data from the least significant packed data element positions of the first signed source 1001 and the second unsigned source 1003 are multiplied using a multiplier circuit 1011. In some embodiments, the signed packed data elements of the first signed source 1001 are sign extended and the unsigned packed data elements of the second unsigned source 1003 are zero extended prior to the multiplications.

[0067] In some embodiments, these multiplier circuits 1005-1011 are reused for other packed data elements positions. In other embodiments, additional multiplier circuits are used so that the packed data elements are processed in parallel. In some contexts, parallel execution is done using lanes that are the size of third signed source 1015 (initial or previous result). The result of the addition of the results of the multiplications is added to the data from most significant packed data element position of third signed source 1015 (initial or previous result) using adder/saturation 1013 circuitry.

[0068] Addition/saturation (accumulator) circuitry 1013 preserves a sign of an operand when the addition results in a value that is too big or too small for signed saturation. In particular, saturation evaluation occurs on the infinite precision result between the multi-way-add and the write to the destination. When the accumulator 1013 is floating-point and the input terms are integer, the sum of products and the floating-point accumulator input value are turned into infinite precision values (fixed point numbers of hundreds of bits), the addition of the multiplication results and the third input is performed, and a single rounding to the actual accumulator type is performed.

[0069] The result 1019 of the addition and saturation check is stored into the signed destination in a packed data element position that corresponds to the packed data element position used from third signed source 1015 (initial or previous result) or passed to the next iteration. In some embodiments, a writemask is applied to this storage such that if a corresponding writemask (bit) is set, the storage happens, and, if not set, the storage does not happen.

[0070] **Figure 11** illustrates power-of-two sized SIMD implementations wherein the accumulators use input sizes that are larger than the inputs to the multipliers according to an embodiment. Note the source (to the multipliers) and accumulator values may be signed or unsigned values. For an accumulator having 2X input sizes (in other words, the accumulator input value is twice the size of the packed data element sizes of the sources), table 1101 illustrates different configurations. For byte sized sources, the accumulator uses word or half-precision floating-point (HPFP) values that are 16-bit in size. For word sized sources, the accumulator uses 32-bit integer or single-precision floating-point (SPFP) values that are 32-bit in size. For SPFP or 32-bit integer sized sources, the accumulator uses 64-intenger or double-precision floating-point (DPFP) values that are 64-bit in size.

[0071] For an accumulator having 4X input sizes (in other words, the accumulator input value is four times the size of the packed data element sizes of the sources), table 1103 illustrates different configurations. For byte sized sources, the accumulator uses 32-bit integer or single-precision floating-point (SPFP) values that are 32-bit in size. For word sized sources, the accumulator uses 64-bit integer or double-precision floating-point (DPFP) values that are 64-bit in size in some embodiments.

[0072] For an accumulator having 8X input sizes (in other words, the accumulator input value is eight times the size of the packed data element sizes of the sources), table 1105 illustrates a configuration. For byte sized sources, the accumulator uses 64-bit integer.

[0073] As hinted at earlier, matrix operations circuitry may be included in a core, or as an external accelerator. **Figure 12** illustrates an embodiment of a system utilizing matrix operations circuitry. In this illustration, multiple entities are coupled with a ring interconnect 1245.

[0074] A plurality of cores, core 0 1201, core 1 1203, core 2 1205, and core N 1207 provide non-tile-based instruction support. In some embodiments, matrix operations circuitry 1251 is provided in a core 1203, and in other embodiments matrix operations circuitry 1211 and 1213 are accessible on the ring interconnect 1245.

[0075] Additionally, one or more memory controllers 1223-1225 are provided to communicate with memory 1233 and 1231 on behalf of the cores and/or matrix operations circuitry.

[0076] **Figure 13** illustrates an embodiment of a processor core pipeline supporting matrix operations using tiles. Branch prediction and decode circuitry 1303 performs branch predicting of instructions, decoding of instructions, and/or both from instructions stored in instruction storage 1301. For example, instructions detailed herein may be stored in instruction storage. In some implementations, separate circuitry is used for branch prediction and in some embodiments, at least some instructions are decoded into one or more micro-operations, micro-code entry points, microinstructions, other instructions, or other control signals using microcode 1305. The branch prediction and decode circuitry 1303 may be implemented using various different mechanisms. Examples of suitable mechanisms include, but are not limited to, look-up tables, hardware implementations, programmable logic arrays (PLAs), microcode read only memories (ROMs), etc.

[0077] The branch prediction and decode circuitry 1303 is coupled to allocate/rename 1307 circuitry which is coupled, in some embodiments, to scheduler circuitry 1309. In some embodiments, these circuits provide register renaming, register allocation, and/or scheduling functionality by performing one or more of: 1) renaming logical operand values to physical operand values (e.g., a register alias table in some embodiments), 2) allocating status bits and flags to the decoded instruction, and 3) scheduling the decoded instruction for execution on execution circuitry out of an instruction pool (e.g., using a reservation station in some embodiments).

[0078] The scheduler circuitry 1309 represents any number of different schedulers, including reservations stations, central instruction window, etc. The scheduler circuitry 1309 is coupled to, or includes, physical register file(s) 1315. Each of the physical register file(s) 1315 represents one or more physical register files, different ones of which store one or more different data types, such as scalar integer, scalar floating-point, packed integer, packed floating-point, vector integer, vector floating-point, status (e.g., an instruction pointer that is the address of the next instruction to be executed), tiles, etc. In one embodiment, the physical register file(s) 1315 comprises vector registers circuitry, write mask registers circuitry, and scalar registers circuitry. These register circuits may provide architectural vector registers, vector mask registers, and general-purpose registers. The physical register file(s) 1315 is overlapped by a retirement circuit 1317 to illustrate various ways in which register renaming and out-of-order execution may be implemented (e.g., using a reorder buffer(s) and a retirement register file(s); using a future file(s), a history buffer(s), and a retirement register file(s); using a register maps and a pool of registers; etc.). The retirement circuit 1317 and the physical register file(s) 1315 are coupled to the execution circuitry 1311.

[0079] While register renaming is described in the context of out-of-order execution, it should be understood that register renaming may be used in an in-order architecture. While the illustrated embodiment of the processor may also include separate instruction and data cache units and a shared L2 cache unit, alternative embodiments may have a single internal cache for both instructions and data, such as, for example, a Level 1 (L1) internal cache, or multiple levels of internal cache. In some embodiments, the system may include a combination of an internal cache and an external cache that is external to the core and/or the processor. Alternatively, all of the cache may be external to the core and/or the processor.

[0080] The execution circuitry 1311 is a set of one or more execution circuits, including scalar circuitry 1321, vector/SIMD circuitry 1323, and matrix operations circuitry 1327, as well as memory access circuitry 1325 to access cache 1313. The execution circuits perform various operations (e.g., shifts, addition, subtraction, multiplication) and on various types of data (e.g., scalar floating-point, packed integer, packed floating-point, vector integer, vector floating-point). While some embodiments may include a number of execution units dedicated to specific functions or sets of functions, other embodiments may include only one execution unit or multiple execution units that all perform all functions. The scalar circuitry 1321 performs scalar operations, the vector/SIMD circuitry 1323 performs vector/SIMD operations, and matrix operations circuitry 1327 performs matrix (tile) operations detailed herein.

[0081] By way of example, the exemplary register renaming, out-of-order issue/execution core architecture may implement a pipeline as follows: 1) an instruction fetch circuit performs fetch and length decoding stages; 2) the branch and decode circuitry 1303 performs a decode stage; 3) the allocate/rename 1307 circuitry performs an allocation stage and renaming stage; 4) the scheduler circuitry 1309 performs a schedule stage; 5) physical register file(s) (coupled to, or included in, the scheduler circuitry 1309 and allocate/rename 1307 circuitry and a memory unit perform a register read/memory read stage; the execution circuitry 1311 performs an execute stage; 6) a memory unit and the physical register file(s) unit(s) perform a write back/memory write stage; 7) various units may be involved in the exception handling stage; and 8) a retirement unit and the physical register file(s) unit(s) perform a commit stage.

[0082] The core may support one or more instructions sets (e.g., the x86 instruction set (with some extensions that have been added with newer versions); the MIPS instruction set of MIPS Technologies of Sunnyvale, CA; the ARM instruction set (with optional additional extensions such as NEON) of ARM Holdings of Sunnyvale, CA), including the instruction(s) described herein. In one embodiment, the core 1390 includes logic to support a packed data instruction set extension (e.g., AVX1, AVX2), thereby allowing the operations used by many multimedia applications to be performed using packed data.

[0083] It should be understood that the core may support multithreading (executing two or more parallel sets of operations or threads), and may do so in a variety of ways including time sliced multithreading, simultaneous multithreading (where a single physical core provides a logical core for each of the threads that physical core is simultaneously multithreading), or a combination thereof (e.g., time sliced fetching and decoding and simultaneous multithreading thereafter such as in the Intel® Hyperthreading technology).

[0084] **Figure 14** illustrates an embodiment of a processor core pipeline supporting matrix operations using tiles. Branch prediction and decode circuitry 1403 performs branch predicting of instructions, decoding of instructions, and/or both from instructions stored in instruction storage 1401. For example, instructions detailed herein may be stored in instruction storage. In some implementations, separate circuitry is used for branch prediction and in some embodiments, at least some instructions are decoded into one or more micro-operations, micro-code entry points, microinstructions, other instructions, or other control signals using microcode 1405. The branch prediction and decode circuitry 1403 may be implemented using various different mechanisms. Examples of suitable mechanisms include, but are not limited to, look-up tables, hardware implementations, programmable logic arrays (PLAs), microcode read only memories (ROMs), etc.

[0085] The branch prediction and decode circuitry 1403 is coupled to allocate/rename 1407 circuitry which is coupled, in some embodiments, to scheduler circuitry 1409. In some embodiments, these circuits provide register renaming, register allocation, and/or scheduling functionality by performing one or more of: 1) renaming logical operand values to physical operand values (e.g., a register alias table in some embodiments), 2) allocating status bits and flags to the decoded instruction, and 3) scheduling the decoded instruction for execution on execution circuitry out of an instruction pool (e.g., using a reservation station in some embodiments).

[0086] The scheduler circuitry 1409 represents any number of different schedulers, including reservations stations, central instruction window, etc. The scheduler unit(s) scheduler circuitry 1409 is coupled to, or includes, physical register file(s) 1415. Each of the physical register file(s) 1415 represents one or more physical register files, different ones of which store one or more different data types, such as scalar integer, scalar floating-point, packed integer, packed floating-point, vector integer, vector floating-point, status (e.g., an instruction pointer that is the address of the next instruction to be executed), tiles, etc. In one embodiment, the physical register file(s) 1415 comprises vector registers circuitry, write mask registers circuitry, and scalar registers circuitry. These register circuits may provide architectural vector registers, vector mask registers, and general-purpose registers. The physical register file(s) 1415 is overlapped by a retirement circuit 1417 to illustrate various ways in which register renaming and out-of-order execution may be implemented (e.g., using a reorder buffer(s) and a retirement register file(s); using a future file(s), a history buffer(s), and a retirement register file(s); using a register maps and a pool of registers; etc.). The retirement circuit 1417 and the physical register file(s) 1415 are coupled to the execution circuitry 1411.

[0087] While register renaming is described in the context of out-of-order execution, it should be understood that register renaming may be used in an in-order architecture. While the illustrated embodiment of the processor may also include separate instruction and data cache units and a shared L2 cache unit, alternative embodiments may have a single internal cache for both instructions and data, such as, for example, a Level 1 (L1) internal cache, or multiple levels of internal cache. In some embodiments, the system may include a combination of an internal cache and an external cache that is external to the core and/or the processor. Alternatively, all of the cache may be external to the core and/or the processor.

[0088] The execution circuitry 1411 a set of one or more execution circuits 1427 and a set of one or more memory access circuits 1425 to access cache 1413. The execution circuits 1427 perform matrix (tile) operations detailed herein.

[0089] By way of example, the exemplary register renaming, out-of-order issue/execution core architecture may implement a pipeline as follows: 1) an instruction fetch circuit performs fetch and length decoding stages; 2) the branch and decode circuitry 1403 performs a decode stage; 3) the allocate/rename 1407 circuitry performs an allocation stage and renaming stage; 4) the scheduler circuitry 1409 performs a schedule stage; 5) physical register file(s) (coupled to, or included in, the scheduler circuitry 1409 and allocate/rename 1407 circuitry and a memory unit perform a register read/memory read stage; the execution circuitry 1411 performs an execute stage; 6) a memory unit and the physical register file(s) unit(s) perform a write back/memory write stage; 7) various units may be involved in the exception handling stage; and 8) a retirement unit and the physical register file(s) unit(s) perform a commit stage.

[0090] The core may support one or more instructions sets (e.g., the x86 instruction set (with some extensions that have been added with newer versions); the MIPS instruction set of MIPS Technologies of Sunnyvale, CA; the ARM instruction set (with optional additional extensions such as NEON) of ARM Holdings of Sunnyvale, CA), including the instruction(s) described herein. In one embodiment, the core 1490 includes logic to support a packed data instruction set extension (e.g., AVX1, AVX2), thereby allowing the operations used by many multimedia applications to be performed using packed data.

[0091] It should be understood that the core may support multithreading (executing two or more parallel sets of operations or threads), and may do so in a variety of ways including time sliced multithreading, simultaneous multithreading (where a single physical core provides a logical core for each of the threads that physical core is simultaneously multithreading), or a combination thereof (e.g., time sliced fetching and decoding and simultaneous multithreading thereafter such as in the Intel® Hyperthreading technology).

LAYOUT

[0092] Throughout this description, data is expressed using row major data layout. Column major users should translate the terms according to their orientation. **Figure 15** illustrates an example of a matrix expressed in row major format and column major format. As shown, matrix A is a 2x3 matrix. When this matrix is stored in row major format, the data elements of a row are consecutive. When this matrix is stored in column major format, the data elements of a column are consecutive. It is a well-known property of matrices that $A^T * B^T = (BA)^T$, where superscript T means transform. Reading column major data as row major data results in the matrix looking like the transform matrix.

[0093] In some embodiments, row-major semantics are utilized in hardware, and column major data is to swap the operand order with the result being transforms of matrix, but for subsequent column-major reads from memory it is the correct, non-transformed matrix.

[0094] For example, if there are two column-major matrices to multiply:

$$\begin{array}{lll} a\ b & g\ i\ k & ag+bh\ ai+bj\ ak+bl \\ c\ d\ * & h\ j\ l = & cg+dh\ ci+dj\ ck+dl \\ e\ f & & eg+fh\ ei+fj\ ek+fl \\ (3x2) & (2x3) & (3x3) \end{array}$$

[0095] The input matrices would be stored in linear memory (column-major) as:

acebdf
and
g h i j k l.

[0096] Reading those matrices as row-major with dimensions 2x3 and 3x2, they would appear as:

$$\begin{array}{lll} ace & \text{and} & g\ h \\ b\ d\ f & & i\ j \\ kl & & \end{array}$$

[0097] Swapping the order and matrix multiplying:

$$\begin{array}{lll} g\ h & a\ c\ e & ag+bh\ cg+dh\ eg+fh \\ i\ j & *\ b\ d\ f = & ai+bj\ ci+dj\ ei+fj \\ k\ l & & ak+bl\ ck+dl\ ek+fl \end{array}$$

[0098] The transform matrix is out and can then be stored in in row-major order:
ag+bh cg+dh eg+fh ai+bj ci+dj ei+fj ak+bl ck+dl ek+fl

and used in subsequent column major computations, it is the correct un-transformed matrix:

ag+bh	ai+bj	ak+bl
cg+dh	ci+dj	ck+dl
eg+fh	ei+fj	ek+fl

EXEMPLARY USAGE

[0099] **Figure 16** illustrates an example of usage of matrices (e.g., tiles). In this example, matrix C 1601 includes two tiles, matrix A 1603 includes one tile, and matrix B 1605 includes two tiles. This figure shows an example of the inner loop of an algorithm to compute a matrix multiplication. In this example, two result tiles, tmm0 and tmm1, from matrix C 1601 are used to accumulate the intermediate results. One tile from the matrix A 1603 (tmm2) is re-used twice as it multiplied by two tiles from matrix B 1605. Pointers to load a new A matrix (tile) and two new B matrices (e.g., tiles) from the directions indicated by the arrows. An outer loop, not shown, adjusts the pointers for the C tiles.

[0100] The exemplary code as shown includes the usage of a tile configuration instruction and is executed to configure tile usage, load tiles, a loop to process the tiles, store tiles to memory, and release tile usage.

[0101] **Figure 17** illustrates an embodiment of usage of matrices (e.g., tiles). At 1701, tile usage is configured. For example, a TILECONFIG instruction is executed to configure tile usage including setting a number of rows and columns per tile. Typically, at least one matrix (tile) is loaded from memory at 1703. At least one matrix (tile) operation is performed at 1705 using the matrices (e.g., tiles). At 1707, at least one matrix (tile) is stored out to memory and a context switch can occur at 1709.

EXEMPLARY CONFIGURATION

TILE CONFIGURATION HARDWARE SUPPORT

[0102] As discussed above, tile usage typically needs to be configured prior to use. For example, full usage of all rows and columns may not be needed. Not only does not configuring these rows and columns save power in some embodiments, but the configuration may be used to determine if an operation will generate an error. For example, a matrix multiplication of the form (N × M) * (L × N) will typically not work if M and L are not the same.

[0103] Prior to using matrices using tiles, in some embodiments, tile support is to be configured. For example, how many rows and columns per tile, tiles that are to be used, etc. are configured. A TILECONFIG instruction is an improvement to a computer itself as it provides for support to configure the computer to use a matrix accelerator (either as a part of a processor core, or as an external device). In particular, an execution of the TILECONFIG instruction causes a configuration to be retrieved from memory and applied to matrix (tile) settings within a matrix accelerator.

TILE USAGE CONFIGURATION

[0104] **Figure 18** illustrates support for configuration of the usage of tiles according to an embodiment. A memory 1801 contains the tile description 1803 of the matrices (e.g., tiles) to be supported.

[0105] Instruction execution resources 1811 of a processor/core 1805 stores aspects of a tile description 1803 into tile configurations 1817. The tile configurations 1817 include palette table 1813 to detail what tiles for a palette are configured (the number of rows and columns in each tile) and a marking that matrix support is in use. In particular, instruction execution resources 1811 are configured to use tiles as specified by the tile configurations 1817. The instruction execution resources 1811 may also include a machine specific register or configuration register to indicate tile usage. Additional values such as in-use and start values are also set. The tile configurations 1817 utilize register(s) 1819 to store tile usage and configuration information.

[0106] **Figure 19** illustrates an embodiment of a description of the matrices (e.g., tiles) to be supported. This is the description that is to be stored upon an execution of a STTILECFG instruction. In this example, each field is a byte. In byte [0], a palette ID 1901 is stored. The palette ID is used to index a palette table 1813 which stores, per palette ID, a number of bytes in a tile, and bytes per row of the tiles that are associated with this ID as defined by the configuration.

[0107] Byte 1 stores a value to be stored in a "startRow" register 1903 and byte 2 stores a value to be stored in a register, startP 1905. To support restarting instructions after these events, the instructions store information these registers. To support restarting instructions after break events such as those detailed above, the instructions store information in these registers. The startRow value indicates the row that should be used for restart. The startP value indicates the position within the row for store operations when pairs are used and, in some embodiments, indicates the lower half of the row (in the lower tile of a pair) or higher half of the row (in the higher tile of a pair). Generally, the position in the row (the column) is not needed.

[0108] With the exception of TILECONFIG and STTILECFG, successfully executing matrix (tile) instructions will set both startRow and startP to zero.

[0109] Any time an interrupted matrix (tile) instruction is not restarted, it is the responsibility of software to zero the startRow and startP values. For example, unmasked floating-point exception handlers might decide to finish the operation in software and change the program counter value to another instruction, usually the next instruction. In this case the software exception handler must zero the startRow and startP values in the exception presented to it by the operating system before resuming the program. The operating system will subsequently reload those values using a restore instruction.

[0110] Byte 3 stores an indication of pairs (1b per tile) of tiles 1907.

[0111] Bytes 16-17 store the number of rows 1913 and columns 1915 for tile 0, bytes 18-19 store the number of rows and columns for tile 1, etc. In other words, each 2-byte group specifies a number of rows and columns for a tile. If a group of 2 bytes is not used to specify tile parameters, they should have the value zero. Specifying tile parameters for more tiles than the implementation limit or the palette limit results in a fault. Unconfigured tiles are set to an initial state with 0 rows, 0 columns.

[0112] Finally, the configuration in memory typically ends with an ending delineation such as all zeros for several consecutive bytes.

EXEMPLARY TILE AND TILE CONFIGURATION STORAGE

[0113] **Figures 20(A)-(D)** illustrate examples of register(s) 1819. **Figure 20(A)** illustrates a plurality of registers 1819. As shown each tile (TMM0 2001 ... TMMN 2003) has a separate register with each register storing a row and column size for that particular tile. StartP 2011 and StartRow 2013 are stored in separate registers. One or more status registers 2015 are set (e.g., TILES_CONFigureD = 1) to indicate tiles are configured for use.

[0114] **Figure 20(B)** illustrates a plurality of registers 1819. As shown each tile has separate registers for its rows and columns. For example, TMM0 rows configuration 2021, TMM0 columns configuration 2023, StartP 2011 and StartRow 2013 are stored in separate registers. One or more status registers 2015 are set (e.g., TILES_CONFigureD = 1) to indicate tiles are configured for use.

[0115] **Figure 20(C)** illustrates a single register 1819. As shown, this register stores tile configurations (rows and columns per tile) 2031, StartP 2011, and StartRow 2013 are stored in single register as packed data registers. One or more status registers 2015 are set (e.g., TILES_CONFigureD = 1) to indicate tiles are configured for use.

[0116] **Figure 20(D)** illustrates a plurality of registers 1819. As shown, a single register stores tile configuration (rows and columns per tile) 2031. StartP and StartRow are stored in separate registers 2011 and 2013. One or more status registers 2015 are set (e.g., TILES_CONFigureD = 1) to indicate tiles are configured for use.

[0117] Other combinations are contemplated such as combining the start registers into a single register where they are shown separately, etc.

[0118] **Figure 21** illustrates an exemplary execution of a tile matrix multiplication on FP19 data elements instruction according to some embodiments. The tile matrix multiplication on FP19 data elements instruction (herein TMMULFP19PS) includes fields for an opcode, an identifier of a location of a first source operand (TSRC1 2101), an identifier of a location of a second source operand (TSRC1 2103), and an identifier of a location of a destination/source operand (TSRCDEST 2131). As such, the operands are tiles (matrices). The locations of the matrices may be provided by a logical register identifier or a memory address. In some embodiments, the tiles are physically a collection of registers with a tile overlay.

[0119] As shown, the source operands 2101 and 2103 store single precision floating point (FP32) data elements.

[0120] The source operands 2101 and 2103 are fed into execution circuitry 2109 to be operated on. In particular, in some embodiments, execution circuitry 2109 performs a signaling not-a-number (SNAN) to quiet NAN (QNAN) conversion of the data elements of the source operands 2101 and 2103 using SNAN to QNAN circuitry 2111, converts the QNAN data elements from a FP32 format to a floating point 19 (FP19) format having 1 sign bit, 8 exponent bits, and 10 mantissa bits that are the corresponding most significant bits of a single precision floating number using FP conversion circuitry 2113, and performs matrix multiplication on the FP19 data elements and accumulates the results with the data elements of the source/destination 2131 in FP32 format using multiplication and accumulation circuitry 2115.

[0121] In some embodiments, the SNAN to QNAN circuitry 2111 generates a QNAN value by determining the exponent is in unsigned form and has a value of 255, that the fraction is a non-zero, and that bit 22 of the fraction is 0, and sets bit 22 of the fraction to be 1.

[0122] In some embodiments, the FP conversion circuitry 2113 generates a FP19 value by setting the lower 13 bits of the FP32 value to be zero.

[0123] **Figure 22** illustrates an embodiment of method to process a TMMULFP19PS instruction. For example, a processor core as shown in Figure 32(B), a pipeline as detailed below, etc. performs this method. In some embodiments, the processor core includes an instruction translator to perform an instruction set architecture (ISA) translation. In other embodiments, a software based translation is used.

[0124] At 2201, a single instruction is fetched. In some embodiments, the single instruction includes fields for an opcode, an indication of a location of a first source operand, an indication of a location of a second source operand, and an indication of a location of a destination operand, wherein the opcode is to indicate that execution circuitry is to at least convert data elements of the first and second source operands from a first floating point representation to a second floating point representation, perform matrix multiplication with the converted data elements, and accumulate results of the matrix multiplication in the destination operand in the first floating point representation.

[0125] In some embodiments, the fetched instruction of the first instruction set architecture is translated into one or more instructions of a second instruction set architecture at 2202.

[0126] The single instruction of the first instruction set architecture or one or more translated instructions of the second instruction set architecture are decoded at 2203.

[0127] Data values associated with the source operands of the decoded instruction are retrieved and the instruction(s) scheduled at 2205. For example, when one or more of the source operands are memory operands, the data from the indicated memory location is retrieved.

[0128] At 2207, the decoded single instruction of the first instruction set architecture or the decoded instruction(s) of the second instruction set is/are executed by execution circuitry (hardware) such as that detailed herein to perform the operations indicated by the opcode of the single first instruction of the first instruction set architecture. For the TMMULFP19PS instruction, the execution will cause execution circuitry to at least convert data elements of the first and second source operands from a first floating point representation to a second floating point representation, perform matrix multiplication with the converted data elements, and accumulate results of the matrix multiplication in the destination operand in the first floating point representation. Note that in most embodiments the first floating point representation is FP32 and the second floating point representation is FP19, however, in some embodiments other representations are used. Further, in some embodiments, the opcode of the single instruction of the first ISA (and subsequent operations to be performed) include one or more of converting from SNAN to QNAN. In some embodiments, remaining data elements of the destination operand are zeroed.

[0129] In some embodiments, the instruction is committed or retired at 2209.

[0130] **Figure 23** illustrates embodiments of execution of a TMMULFP19PS instruction. In particular, the execution of 2207.

[0131] For each indicated data element of the first source and the second source operands, the data elements are converted from SNAN to QNAN at 2301. Note that when a data element is not in SNAN format, there is no conversion.

[0132] The (in some embodiments QNAN) data elements of the first and second source operands are converted from FP32 to FP19 by zeroing the lower order mantissa bits of each converted data element to generate a value to be multiplied at 2303.

[0133] Matrix multiplication in FP32 for each of the generated values to be multiplied is performed at 2305. The multiplied values are accumulated in the destination operand in FP32 format at 2307 and remaining data elements of the destination operand are zeroed at 2307.

[0134] **Figure 24** illustrates embodiments pseudo-code for the execution of TMMULFP19PS instruction.

[0135] **Figure 25** illustrates an exemplary execution of a tile matrix transpose and multiplication on FP19 data elements instruction according to some embodiments. The tile matrix transpose and multiplication on FP19 data elements instruction (herein TTMULFP19PS) includes fields for an opcode, an identifier of a location of a first source operand (TSRC1 2501), an identifier of a location of a second source operand (TSRC1 2503), and an identifier of a location of a destination/source operand (TSRCDEST 2531). As such, the operands are tiles (matrices). The locations of the matrices may be provided by a logical register identifier or a memory address. In some embodiments, the tiles are physically a collection of registers with a tile overlay.

[0136] As shown, the source operands 2501 and 2503 store single precision floating point (FP32) data elements.

[0137] The source operands 2501 and 2503 are fed into execution circuitry 2509 to be operated on. In particular, in some embodiments, execution circuitry 2509 performs a signaling not-a-number (SNAN) to quiet NAN (QNAN) conversion of the data elements of the source operands 2501 and 2503 using SNAN to QNAN circuitry 2511, converts the QNAN data elements from a FP32 format to a floating point 19 (FP19) format having 1 sign bit, 8 exponent bits, and 10 mantissa bits using FP conversion circuitry 2513, transposes the FP19 formatted data elements of the first source using transpose circuitry 2512, and performs matrix multiplication on the FP19 data elements of the second source and the transposed FP19 data elements of the first source, and accumulates the results with the data elements of the source/destination 2531 in FP32 format using multiplication and accumulation circuitry 2515. Note that the transpose of the first source may occur at any stage before the multiplication and accumulation.

[0138] In some embodiments, the SNAN to QNAN circuitry 2511 generates a QNAN value by determining the exponent is in unsigned form and has a value of 255, that the fraction is a non-zero, and that bit 22 of the fraction is 0, and sets bit 22 of the fraction to be 1.

[0139] In some embodiments, the FP conversion circuitry 2513 generates a FP19 value by setting the lower 13 bits of the FP32 value to be zero.

[0140] **Figure 26** illustrates an embodiment of method to process a TTMULFP19PS instruction. For example, a processor core as shown in Figure 32(B), a pipeline as detailed below, etc. performs this method. In some embodiments, the processor core includes an instruction translator to perform an instruction set architecture (ISA) translation. In other embodiments, a software based translation is used.

[0141] At 2601, a single instruction is fetched. In some embodiments, the single instruction includes fields for an opcode, an indication of a location of a first source operand, an indication of a location of a second source operand, and an indication of a location of a destination operand, wherein the opcode is to indicate that execution circuitry is to at least convert data elements of the first and second source operands from a first floating point representation to a second floating point representation, perform matrix multiplication with a transpose of the converted data elements of the first source operand and the non-transposed, converted data elements of the second source operand, and accumulate results of the matrix multiplication in the destination operand in the first floating point representation..

[0142] In some embodiments, the fetched instruction of the first instruction set architecture is translated into one or more instructions of a second instruction set architecture at 2602.

[0143] The single instruction of the first instruction set architecture or one or more translated instructions of the second instruction set architecture are decoded at 2603.

[0144] Data values associated with the source operands of the decoded instruction are retrieved and the instruction(s) scheduled at 2605. For example, when one or more of the source operands are memory operands, the data from the indicated memory location is retrieved.

[0145] At 2607, the decoded single instruction of the first instruction set architecture or the decoded instruction(s) of the second instruction set is/are executed by execution circuitry (hardware) such as that detailed herein to perform the operations indicated by the opcode of the single first instruction of the first instruction set architecture. For the TTMULFP19PS instruction, the execution will cause execution circuitry to at least convert data elements of the first and second source operands from a first floating point representation to a second floating point representation, transpose the converted data elements of the first source operand, perform matrix multiplication with the converted data elements of the second source operand and the transposed, converted data elements of the second operand, and accumulate results of the matrix multiplication in the destination operand in the first floating point representation. Note that in most embodiments the first floating point representation is FP32 and the second floating point representation is FP19, however, in some embodiments other representations are used. Further, in some embodiments, the opcode of the single instruction of the first ISA (and subsequent operations to be performed) include one or more of converting from SNAN to QNAN. In some embodiments, remaining data elements of the destination operand are zeroed.

[0146] In some embodiments, the instruction is committed or retired at 2609.

[0147] Figure 27 illustrates embodiments of execution of a TTMULFP19PS instruction. In particular, the execution of 2607.

[0148] For each indicated data element of the first source and the second source operands, the data elements are converted from SNAN to QNAN at 2701. Note that when a data element is not in SNAN format, there is no conversion.

[0149] The (in some embodiments QNAN) data elements of the first and second source operands are converted from FP32 to FP19 by zeroing the lower order mantissa bits of each converted data element to generate a value to be multiplied at 2703.

[0150] At 2704 the converted data elements of the first source operand are transposed.

[0151] Matrix multiplication in FP32 for each of the generated or transposed values to be multiplied is performed at 2705. The multiplied values are accumulated in the destination operand in FP32 format at 2707 and remaining data elements of the destination operand are zeroed at 2707.

[0152] **Figure 28** illustrates embodiments of pseudo-code for the execution of a TTMULFP19PS instruction.

[0153] **Figure 29** illustrates embodiments of hardware to process an instruction such as a TTMULFP19PS and/or TMMULFP19PS instruction. As illustrated, storage 2903 stores a TTMULFP19PS and/or TMMULFP19PS instruction 2901 to be executed.

[0154] The instruction 2901 is received by decode circuitry 2905. For example, the decode circuitry 2905 receives this instruction from fetch logic/circuitry. The instruction includes fields for an opcode, first and second sources, and a source/destination. In some embodiments, the sources and source/destination are tiles (e.g., stored in registers, and in other embodiments, stored in one or more are memory locations).

[0155] More detailed embodiments of at least one instruction format will be detailed later. The decode circuitry 2905 decodes the instruction into one or more operations. In some embodiments, this decoding includes generating a plurality of micro-operations to be performed by execution circuitry (such as execution circuitry 2909). The decode circuitry 2905 also decodes instruction prefixes.

[0156] In some embodiments, register renaming, register allocation, and/or scheduling circuitry 2907 provides functionality for one or more of: 1) renaming logical operand values to physical operand values (e.g., a register alias table in some embodiments), 2) allocating status bits and flags to the decoded instruction, and 3) scheduling the decoded instruction for execution on execution circuitry out of an instruction pool (e.g., using a reservation station in some embodiments).

[0157] Registers (register file) and/or memory 2908 store data as operands of the instruction to be operated on by execution circuitry 2909. Exemplary register types include packed data registers, general purpose registers, and floating-point registers.

[0158] Execution circuitry 2909 executes the decoded instruction. Exemplary detailed execution circuitry is shown in FIGS. 21, 32, 3 (e.g., matrix operations accelerator 307), 4, 5, 6, etc. The execution of the decoded instruction causes the execution circuitry to perform the operations indicated by the opcode. In some embodiments, one or more commands are sent from a processor to an accelerator which then performs the operations dictated by the opcode.

[0159] In some embodiments, retirement/write back circuitry 2911 architecturally commits the destination register into the registers or memory 2908 and retires the instruction.

[0160] An embodiment of a format for an arithmetic recurrence instruction is TTMULFP19PS TSRCDEST, TSRC1, TSRC2. In some embodiments, TTMULFP19PS is the opcode mnemonic of the instruction. TSRCDEST is defined by one or more fields for a tile register operand. TSRC1 and TSRC2 are defined by fields for the sources such as tile registers and/or memory.

[0161] An embodiment of a format for an arithmetic recurrence instruction is TMMULFP19PS TSRCDEST, TSRC1, TSRC2. In some embodiments, TMMULFP19PS is the opcode mnemonic of the instruction. TSRCDEST is defined by one or more fields for a tile register operand. TSRC1 and TSRC2 are defined by fields for the sources such as tile registers and/or memory.

[0162] In some embodiments, theses instructions are in the second format detailed below. In some embodiments, this format is known as VEX.

Exemplary Computer Architectures

[0163] Detailed below are describes of exemplary computer architectures which may be utilized to implement one or more of the above embodied instructions. Other system designs and configurations known in the arts for laptops, desktops, handheld PCs, personal digital assistants, engineering workstations, servers, network devices, network hubs, switches, embedded processors, digital signal processors (DSPs), graphics devices, video game devices, set-top boxes, micro controllers, cell phones, portable media players, hand held devices, and various other electronic devices, are also suitable. In general, a huge variety of systems or electronic devices capable of incorporating a processor and/or other execution logic as disclosed herein are generally suitable.

[0164] **Figure 30** illustrates embodiments of an exemplary system. Multiprocessor system 3000 is a point-to-point interconnect system and includes a plurality of processors including a first processor 3070 and a second processor 3080 coupled via a point-to-point interconnect 3050. In some embodiments, the first processor 3070 and the second processor 3080 are homogeneous. In some embodiments, first processor 3070 and the second processor 3080 are heterogenous.

[0165] Processors 3070 and 3080 are shown including integrated memory controller (IMC) units circuitry 3072 and 3082, respectively. Processor 3070 also includes as part of its interconnect controller units point-to-point (P-P) interfaces 3076 and 3078; similarly, second processor 3080 includes P-P interfaces 3086 and 3088. Processors 3070, 3080 may exchange information via the point-to-point (P-P) interconnect 3050 using P-P interface circuits 3078, 3088. IMCs 3072 and 3082 couple the processors 3070, 3080 to respective memories, namely a memory 3032 and a memory 3034, which may be portions of main memory locally attached to the respective processors.

[0166] Processors 3070, 3080 may each exchange information with a chipset 3090 via individual P-P interconnects 3052, 3054 using point to point interface circuits 3076, 3094, 3086, 3098. Chipset 3090 may optionally exchange information with a coprocessor 3038 via a high-performance interface 3092. In some embodiments, the coprocessor 3038 is a special-purpose processor, such as, for example, a high-throughput MIC processor, a network or communication processor, compression engine, graphics processor, GPGPU, embedded processor, or the like.

[0167] A shared cache (not shown) may be included in either processor 3070, 3080 or outside of both processors, yet connected with the processors via P-P interconnect, such that either or both processors' local cache information may be stored in the shared cache if a processor is placed into a low power mode.

[0168] Chipset 3090 may be coupled to a first interconnect 3016 via an interface 3096. In some embodiments, first interconnect 3016 may be a Peripheral Component Interconnect (PCI) interconnect, or an interconnect such as a PCI Express interconnect or another I/O interconnect. In some embodiments, one of the interconnects couples to a power control unit (PCU) 3017, which may include circuitry, software, and/or firmware to perform power management operations with regard to the processors 3070, 3080 and/or co-processor 3038. PCU 3017 provides control information to a voltage regulator to cause the voltage regulator to generate the appropriate regulated voltage. PCU 3017 also provides control information to control the operating voltage generated. In various embodiments, PCU 3017 may include a variety of power management logic units (circuitry) to perform hardware-based power management. Such power management may be wholly processor controlled (e.g., by various processor hardware, and which may be triggered by workload and/or power, thermal or other processor constraints) and/or the power management may be performed responsive to external sources (such as a platform or power management source or system software).

[0169] PCU 3017 is illustrated as being present as logic separate from the processor 3070 and/or processor 3080. In other cases, PCU 3017 may execute on a given one or more of cores (not shown) of processor 3070 or 3080. In some cases, PCU 3017 may be implemented as a microcontroller (dedicated or general-purpose) or other control logic configured to execute its own dedicated power management code, sometimes referred to as P-code. In yet other embodiments, power management operations to be performed by PCU 3017 may be implemented externally to a processor, such as by way of a separate power management integrated circuit (PMIC) or another component external to the processor. In yet other embodiments, power management operations to be performed by PCU 3017 may be implemented within BIOS or other system software.

[0170] Various I/O devices 3014 may be coupled to first interconnect 3016, along with an interconnect (bus) bridge 3018 which couples first interconnect 3016 to a second interconnect 3020. In some embodiments, one or more additional processor(s) 3015, such as coprocessors, high-throughput MIC processors, GPGPU's, accelerators (such as, e.g., graphics accelerators or digital signal processing (DSP) units), field programmable gate arrays (FPGAs), or any other processor, are coupled to first interconnect 3016. In some embodiments, second interconnect 3020 may be a low pin count (LPC) interconnect. Various devices may be coupled to second interconnect 3020 including, for example, a keyboard and/or mouse 3022, communication devices 3027 and a storage unit circuitry 3028. Storage unit circuitry 3028 may be a disk drive or other mass storage device which may include instructions/code and data 3030, in some embodiments. Further, an audio I/O 3024 may be coupled to second interconnect 3020. Note that other architectures than the point-to-point architecture described above are possible. For example, instead of the point-to-point architecture, a system such as multiprocessor system 3000 may implement a multi-drop interconnect or other such architecture.

Exemplary Core Architectures, Processors, and Computer Architectures

[0171] Processor cores may be implemented in different ways, for different purposes, and in different processors. For instance, implementations of such cores may include: 1) a general purpose in-order core intended for general-purpose computing; 2) a high performance general purpose out-of-order core intended for general-purpose computing; 3) a special purpose core intended primarily for graphics and/or scientific (throughput) computing. Implementations of different processors may include: 1) a CPU including one or more general purpose in-order cores intended for general-purpose computing and/or one or more general purpose out-of-order cores intended for general-purpose computing; and 2) a coprocessor including one or more special purpose cores intended primarily for graphics and/or scientific (throughput). Such different processors lead to different computer system architectures, which may include: 1) the coprocessor on a separate chip from the CPU; 2) the coprocessor on a separate die in the same package as a CPU; 3) the coprocessor on the same die as a CPU (in which case, such a coprocessor is sometimes referred to as special purpose logic, such as integrated graphics and/or scientific (throughput) logic, or as special purpose cores); and 4) a system on a chip that may include on the same die as the described CPU (sometimes referred to as the application core(s) or application processor(s)), the above described coprocessor, and additional functionality. Exemplary core architectures are described next, followed by descriptions of exemplary processors and computer architectures.

[0172] **Figure 31** illustrates a block diagram of embodiments of a processor 3100 that may have more than one core, may have an integrated memory controller, and may have integrated graphics. The solid lined boxes illustrate a processor 3100 with a single core 3102A, a system agent 3110, a set of one or more interconnect controller units circuitry 3116, while the optional addition of the dashed lined boxes illustrates an alternative processor 3100 with multiple cores 3102(A)-(N), a set of one or more integrated memory controller unit(s) circuitry 3114 in the system agent unit circuitry 3110, and special purpose logic 3108, as well as a set of one or more interconnect controller units circuitry 3116. Note that the processor 3100 may be one of the processors 3070 or 3080, or co-processor 3038 or 3015 of **Figure 30.**

[0173] Thus, different implementations of the processor 3100 may include: 1) a CPU with the special purpose logic 3108 being integrated graphics and/or scientific (throughput) logic (which may include one or more cores, not shown), and the cores 3102(A)-(N) being one or more general purpose cores (e.g., general purpose in-order cores, general purpose out-of-order cores, or a combination of the two); 2) a coprocessor with the cores 3102(A)-(N) being a large number of special purpose cores intended primarily for graphics and/or scientific (throughput); and 3) a coprocessor with the cores 3102(A)-(N) being a large number of general purpose in-order cores. Thus, the processor 3100 may be a general-purpose processor, coprocessor or special-purpose processor, such as, for example, a network or communication processor, compression engine, graphics processor, GPGPU (general purpose graphics processing unit circuitry), a high-throughput many integrated core (MIC) coprocessor (including 30 or more cores), embedded processor, or the like. The processor may be implemented on one or more chips. The processor 3100 may be a part of and/or may be implemented on one or more substrates using any of a number of process technologies, such as, for example, BiCMOS, CMOS, or NMOS.

[0174] A memory hierarchy includes one or more levels of cache unit(s) circuitry 3104(A)-(N) within the cores 3102(A)-(N), a set of one or more shared cache units circuitry 3106, and external memory (not shown) coupled to the set of integrated memory controller units circuitry 3114. The set of one or more shared cache units circuitry 3106 may include one or more mid-level caches, such as level 2 (L2), level 3 (L3), level 4 (L4), or other levels of cache, such as a last level cache (LLC), and/or combinations thereof. While in some embodiments ring-based interconnect network circuitry 3112 interconnects the special purpose logic 3108 (e.g., integrated graphics logic), the set of shared cache units circuitry 3106, and the system agent unit circuitry 3110, alternative embodiments use any number of well-known techniques for interconnecting such units. In some embodiments, coherency is maintained between one or more of the shared cache units circuitry 3106 and cores 3102(A)-(N).

[0175] In some embodiments, one or more of the cores 3102(A)-(N) are capable of multithreading. The system agent unit circuitry 3110 includes those components coordinating and operating cores 3102(A)-(N). The system agent unit circuitry 3110 may include, for example, power control unit (PCU) circuitry and/or display unit circuitry (not shown). The PCU may be or may include logic and components needed for regulating the power state of the cores 3102(A)-(N) and/or the special purpose logic 3108 (e.g., integrated graphics logic). The display unit circuitry is for driving one or more externally connected displays.

[0176] The cores 3102(A)-(N) may be homogenous or heterogeneous in terms of architecture instruction set; that is, two or more of the cores 3102(A)-(N) may be capable of executing the same instruction set, while other cores may be capable of executing only a subset of that instruction set or a different instruction set.

Exemplary Core Architectures

In-order and out-of-order core block diagram

[0177] **Figure 32(A)** is a block diagram illustrating both an exemplary in-order pipeline and an exemplary register renaming, out-of-order issue/execution pipeline according to embodiments of the invention. **Figure 32(B)** is a block diagram illustrating both an exemplary embodiment of an in-order architecture core and an exemplary register renaming, out-of-order issue/execution architecture core to be included in a processor according to embodiments of the invention. The solid lined boxes in **FIGS. 32(A)-(B)** illustrate the in-order pipeline and in-order core, while the optional addition of the dashed lined boxes illustrates the register renaming, out-of-order issue/execution pipeline and core. Given that the in-order aspect is a subset of the out-of-order aspect, the out-of-order aspect will be described.

[0178] In **Figure 32(A),** a processor pipeline 3200 includes a fetch stage 3202, an optional length decode stage 3204, a decode stage 3206, an optional allocation stage 3208, an optional renaming stage 3210, a scheduling (also known as a dispatch or issue) stage 3212, an optional register read/memory read stage 3214, an execute stage 3216, a write back/memory write stage 3218, an optional exception handling stage 3222, and an optional commit stage 3224. One or more operations can be performed in each of these processor pipeline stages. For example, during the fetch stage 3202, one or more instructions are fetched from instruction memory, during the decode stage 3206, the one or more fetched instructions may be decoded, addresses (e.g., load store unit (LSU) addresses) using forwarded register ports may be generated, and branch forwarding (e.g., immediate offset or an link register (LR)) may be performed. In one embodiment, the decode stage 3206 and the register read/memory read stage 3214 may be combined into one pipeline stage. In one embodiment, during the execute stage 3216, the decoded instructions may be executed, LSU address/data pipelining to an Advanced Microcontroller Bus (AHB) interface may be performed, multiply and add operations may be performed, arithmetic operations with branch results may be performed, etc.

[0179] By way of example, the exemplary register renaming, out-of-order issue/execution core architecture may implement the pipeline 3200 as follows: 1) the instruction fetch 3238 performs the fetch and length decoding stages 3202 and 3204; 2) the decode unit circuitry 3240 performs the decode stage 3206; 3) the rename/allocator unit circuitry 3252 performs the allocation stage 3208 and renaming stage 3210; 4) the scheduler unit(s) circuitry 3256 performs the schedule stage 3212; 5) the physical register file(s) unit(s) circuitry 3258 and the memory unit circuitry 3270 perform the register read/memory read stage 3214; the execution cluster 3260 perform the execute stage 3216; 6) the memory unit circuitry 3270 and the physical register file(s) unit(s) circuitry 3258 perform the write back/memory write stage 3218; 7) various units (unit circuitry) may be involved in the exception handling stage 3222; and 8) the retirement unit circuitry 3254 and the physical register file(s) unit(s) circuitry 3258 perform the commit stage 3224.

[0180] **Figure 32(B)** shows processor core 3290 including front-end unit circuitry 3230 coupled to an execution engine unit circuitry 3250, and both are coupled to a memory unit circuitry 3270. The core 3290 may be a reduced instruction set computing (RISC) core, a complex instruction set computing (CISC) core, a very long instruction word (VLIW) core, or a hybrid or alternative core type. As yet another option, the core 3290 may be a special-purpose core, such as, for example, a network or communication core, compression engine, coprocessor core, general purpose computing graphics processing unit (GPGPU) core, graphics core, or the like.

[0181] The front end unit circuitry 3230 may include branch prediction unit circuitry 3232 coupled to an instruction cache unit circuitry 3234, which is coupled to an instruction translation lookaside buffer (TLB) 3236, which is coupled to instruction fetch unit circuitry 3238, which is coupled to decode unit circuitry 3240. In one embodiment, the instruction cache unit circuitry 3234 is included in the memory unit circuitry 3270 rather than the front-end unit circuitry 3230. The decode unit circuitry 3240 (or decoder) may decode instructions, and generate as an output one or more micro-operations, micro-code entry points, microinstructions, other instructions, or other control signals, which are decoded from, or which otherwise reflect, or are derived from, the original instructions. The decode unit circuitry 3240 may further include an address generation unit circuitry (AGU, not shown). In one embodiment, the AGU generates an LSU address using forwarded register ports, and may further perform branch forwarding (e.g., immediate offset branch forwarding, LR register branch forwarding, etc.). The decode unit circuitry 3240 may be implemented using various different mechanisms. Examples of suitable mechanisms include, but are not limited to, look-up tables, hardware implementations, programmable logic arrays (PLAs), microcode read only memories (ROMs), etc. In one embodiment, the core 3290 includes a microcode ROM (not shown) or other medium that stores microcode for certain macroinstructions (e.g., in decode unit circuitry 3240 or otherwise within the front end unit circuitry 3230). In one embodiment, the decode unit circuitry 3240 includes a micro-operation (micro-op) or operation cache (not shown) to hold/cache decoded operations, micro-tags, or micro-operations generated during the decode or other stages of the processor pipeline 3200. The decode unit circuitry 3240 may be coupled to rename/allocator unit circuitry 3252 in the execution engine unit circuitry 3250.

[0182] The execution engine circuitry 3250 includes the rename/allocator unit circuitry 3252 coupled to a retirement unit circuitry 3254 and a set of one or more scheduler(s) circuitry 3256. The scheduler(s) circuitry 3256 represents any number of different schedulers, including reservations stations, central instruction window, etc. In some embodiments, the scheduler(s) circuitry 3256 can include arithmetic logic unit (ALU) scheduler/scheduling circuitry, ALU queues, arithmetic generation unit (AGU) scheduler/scheduling circuitry, AGU queues, etc. The scheduler(s) circuitry 3256 is coupled to the physical register file(s) circuitry 3258. Each of the physical register file(s) circuitry 3258 represents one or more physical register files, different ones of which store one or more different data types, such as scalar integer, scalar floating-point, packed integer, packed floating-point, vector integer, vector floating-point, status (e.g., an instruction pointer that is the address of the next instruction to be executed), etc. In one embodiment, the physical register file(s) unit circuitry 3258 includes vector registers unit circuitry, writemask registers unit circuitry, and scalar register unit circuitry. These register units may provide architectural vector registers, vector mask registers, general-purpose registers, etc. The physical register file(s) unit(s) circuitry 3258 is overlapped by the retirement unit circuitry 3254 (also known as a retire queue or a retirement queue) to illustrate various ways in which register renaming and out-of-order execution may be implemented (e.g., using a reorder buffer(s) (ROB(s)) and a retirement register file(s); using a future file(s), a history buffer(s), and a retirement register file(s); using a register maps and a pool of registers; etc.). The retirement unit circuitry 3254 and the physical register file(s) circuitry 3258 are coupled to the execution cluster(s) 3260. The execution cluster(s) 3260 includes a set of one or more execution units circuitry 3262 and a set of one or more memory access circuitry 3264. The execution units circuitry 3262 may perform various arithmetic, logic, floating-point or other types of operations (e.g., shifts, addition, subtraction, multiplication) and on various types of data (e.g., scalar floating-point, packed integer, packed floating-point, vector integer, vector floating-point). While some embodiments may include a number of execution units or execution unit circuitry dedicated to specific functions or sets of functions, other embodiments may include only one execution unit circuitry or multiple execution units/execution unit circuitry that all perform all functions. The scheduler(s) circuitry 3256, physical register file(s) unit(s) circuitry 3258, and execution cluster(s) 3260 are shown as being possibly plural because certain embodiments create separate pipelines for certain types of data/operations (e.g., a scalar integer pipeline, a scalar floating-point/packed integer/packed floating-point/vector integer/vector floating-point pipeline, and/or a memory access pipeline that each have their own scheduler circuitry, physical register file(s) unit circuitry, and/or execution cluster - and in the case of a separate memory access pipeline, certain embodiments are implemented in which only the execution cluster of this pipeline has the memory access unit(s) circuitry 3264). It should also be understood that where separate pipelines are used, one or more of these pipelines may be out-of-order issue/execution and the rest in-order.

[0183] In some embodiments, the execution engine unit circuitry 3250 may perform load store unit (LSU) address/data pipelining to an Advanced Microcontroller Bus (AHB) interface (not shown), and address phase and writeback, data phase load, store, and branches.

[0184] The set of memory access circuitry 3264 is coupled to the memory unit circuitry 3270, which includes data TLB unit circuitry 3272 coupled to a data cache circuitry 3274 coupled to a level 2 (L2) cache circuitry 3276. In one exemplary embodiment, the memory access units circuitry 3264 may include a load unit circuitry, a store address unit circuit, and a store data unit circuitry, each of which is coupled to the data TLB circuitry 3272 in the memory unit circuitry 3270. The instruction cache circuitry 3234 is further coupled to a level 2 (L2) cache unit circuitry 3276 in the memory unit circuitry 3270. In one embodiment, the instruction cache 3234 and the data cache 3274 are combined into a single instruction and data cache (not shown) in L2 cache unit circuitry 3276, a level 3 (L3) cache unit circuitry (not shown), and/or main memory. The L2 cache unit circuitry 3276 is coupled to one or more other levels of cache and eventually to a main memory.

[0185] The core 3290 may support one or more instructions sets (e.g., the x86 instruction set (with some extensions that have been added with newer versions); the MIPS instruction set; the ARM instruction set (with optional additional extensions such as NEON)), including the instruction(s) described herein. In one embodiment, the core 3290 includes logic to support a packed data instruction set extension (e.g., AVX1, AVX2), thereby allowing the operations used by many multimedia applications to be performed using packed data.

Exemplary Execution Unit(s) Circuitry

[0186] **Figure 33** illustrates embodiments of execution unit(s) circuitry, such as execution unit(s) circuitry 3262 of Figure 32(B). As illustrated, execution unit(s) circuity 3262 may include one or more ALU circuits 3301, vector/SIMD unit circuits 3303, load/store unit circuits 3305, and/or branch/jump unit circuits 3307. ALU circuits 3301 perform integer arithmetic and/or Boolean operations. Vector/SIMD unit circuits 3303 perform vector/SIMD operations on packed data (such as SIMD/vector registers). Load/store unit circuits 3305 execute load and store instructions to load data from memory into registers or store from registers to memory. Load/store unit circuits 3305 may also generate addresses. Branch/jump unit circuits 3307 cause a branch or jump to a memory address depending on the instruction. Floating-point unit (FPU) circuits 3309 perform floating-point arithmetic. The width of the execution unit(s) circuitry 3262 varies depending upon the embodiment and can range from 16-bit to 1,024-bit. In some embodiments, two or more smaller execution units are logically combined to form a larger execution unit (e.g., two 128-bit execution units are logically combined to form a 256-bit execution unit).

Exemplary Register Architecture

[0187] **Figure 34** is a block diagram of a register architecture 3400 according to some embodiments. As illustrated, there are vector/SIMD registers 3410 that vary from 128-bit to 1,024 bits width. In some embodiments, the vector/SIMD registers 3410 are physically 512-bits and, depending upon the mapping, only some of the lower bits are used. For example, in some embodiments, the vector/SIMD registers 3410 are ZMM registers which are 512 bits: the lower 256 bits are used for YMM registers and the lower 128 bits are used for XMM registers. As such, there is an overlay of registers. In some embodiments, a vector length field selects between a maximum length and one or more other shorter lengths, where each such shorter length is half the length of the preceding length. Scalar operations are operations performed on the lowest order data element position in a ZMM/YMM/XMM register; the higher order data element positions are either left the same as they were prior to the instruction or zeroed depending on the embodiment.

[0188] In some embodiments, the register architecture 3400 includes writemask/predicate registers 3415. For example, in some embodiments, there are 8 writemask/predicate registers (sometimes called k0 through k7) that are each 16-bit, 32-bit, 64-bit, or 128-bit in size. Writemask/predicate registers 3415 may allow for merging (e.g., allowing any set of elements in the destination to be protected from updates during the execution of any operation) and/or zeroing (e.g., zeroing vector masks allow any set of elements in the destination to be zeroed during the execution of any operation). In some embodiments, each data element position in a given writemask/predicate register 3415 corresponds to a data element position of the destination. In other embodiments, the writemask/predicate registers 3415 are scalable and consists of a set number of enable bits for a given vector element (e.g., 8 enable bits per 64-bit vector element).

[0189] The register architecture 3400 includes a plurality of general-purpose registers 3425. These registers may be 16-bit, 32-bit, 64-bit, etc. and can be used for scalar operations. In some embodiments, these registers are referenced by the names RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, and R8 through R15.

[0190] In some embodiments, the register architecture 3400 includes scalar floating-point register 3445 which is used for scalar floating-point operations on 32/64/80-bit floating-point data using the x87 instruction set extension or as MMX registers to perform operations on 64-bit packed integer data, as well as to hold operands for some operations performed between the MMX and XMM registers.

[0191] One or more flag registers 3440 (e.g., EFLAGS, RFLAGS, etc.) store status and control information for arithmetic, compare, and system operations. For example, the one or more flag registers 3440 may store condition code information such as carry, parity, auxiliary carry, zero, sign, and overflow. In some embodiments, the one or more flag registers 3440 are called program status and control registers.

[0192] Segment registers 3420 contain segment points for use in accessing memory. In some embodiments, these registers are referenced by the names CS, DS, SS, ES, FS, and GS.

[0193] Machine specific registers (MSRs) 3435 control and report on processor performance. Most MSRs 3435 handle system-related functions and are not accessible to an application program. Machine check registers 3460 consist of control, status, and error reporting MSRs that are used to detect and report on hardware errors.

[0194] One or more instruction pointer register(s) 3430 store an instruction pointer value. Control register(s) 3455 (e.g., CR0-CR4) determine the operating mode of a processor (e.g., processor 3070, 3080, 3038, 3015, and/or 3100) and the characteristics of a currently executing task. Debug registers 3450 control and allow for the monitoring of a processor or core's debugging operations.

[0195] Memory management registers 3465 specify the locations of data structures used in protected mode memory management. These registers may include a GDTR, IDRT, task register, and a LDTR register.

[0196] Alternative embodiments of the invention may use wider or narrower registers. Additionally, alternative embodiments of the invention may use more, less, or different register files and registers.

Instruction Sets

[0197] An instruction set architecture (ISA) may include one or more instruction formats. A given instruction format may define various fields (e.g., number of bits, location of bits) to specify, among other things, the operation to be performed (e.g., opcode) and the operand(s) on which that operation is to be performed and/or other data field(s) (e.g., mask). Some instruction formats are further broken down though the definition of instruction templates (or sub-formats). For example, the instruction templates of a given instruction format may be defined to have different subsets of the instruction format's fields (the included fields are typically in the same order, but at least some have different bit positions because there are less fields included) and/or defined to have a given field interpreted differently. Thus, each instruction of an ISA is expressed using a given instruction format (and, if defined, in a given one of the instruction templates of that instruction format) and includes fields for specifying the operation and the operands. For example, an exemplary ADD instruction has a specific opcode and an instruction format that includes an opcode field to specify that opcode and operand fields to select operands (source1/destination and source2); and an occurrence of this ADD instruction in an instruction stream will have specific contents in the operand fields that select specific operands.

Exemplary Instruction Formats

[0198] Embodiments of the instruction(s) described herein may be embodied in different formats. Additionally, exemplary systems, architectures, and pipelines are detailed below. Embodiments of the instruction(s) may be executed on such systems, architectures, and pipelines, but are not limited to those detailed.

[0199] **Figure 35** illustrates embodiments of an instruction format. As illustrated, an instruction may include multiple components including, but not limited to, one or more fields for: one or more prefixes 3501, an opcode 3503, addressing information 3505 (e.g., register identifiers, memory addressing information, etc.), a displacement value 3507, and/or an immediate 3509. Note that some instructions utilize some or all of the fields of the format whereas others may only use the field for the opcode 3503. In some embodiments, the order illustrated is the order in which these fields are to be encoded, however, it should be appreciated that in other embodiments these fields may be encoded in a different order, combined, etc.

[0200] The prefix(es) field(s) 3501, when used, modifies an instruction. In some embodiments, one or more prefixes are used to repeat string instructions (e.g., 0xF0, 0xF2, 0xF3, etc.), to provide section overrides (e.g., 0x2E, 0x36, 0x3E, 0x26, 0x64, 0x65, 0x2E, 0x3E, etc.), to perform bus lock operations, and/or to change operand (e.g., 0x66) and address sizes (e.g., 0x67). Certain instructions require a mandatory prefix (e.g., 0x66, 0xF2, 0xF3, etc.). Certain of these prefixes may be considered "legacy" prefixes. Other prefixes, one or more examples of which are detailed herein, indicate, and/or provide further capability, such as specifying particular registers, etc. The other prefixes typically follow the "legacy" prefixes.

[0201] The opcode field 3503 is used to at least partially define the operation to be performed upon a decoding of the instruction. In some embodiments, a primary opcode encoded in the opcode field 3503 is 1, 2, or 3 bytes in length. In other embodiments, a primary opcode can be a different length. An additional 3-bit opcode field is sometimes encoded in another field.

[0202] The addressing field 3505 is used to address one or more operands of the instruction, such as a location in memory or one or more registers. **Figure 36** illustrates embodiments of the addressing field 3505. In this illustration, an optional ModR/M byte 3602 and an optional Scale, Index, Base (SIB) byte 3604 are shown. The ModR/M byte 3602 and the SIB byte 3604 are used to encode up to two operands of an instruction, each of which is a direct register or effective memory address. Note that each of these fields are optional in that not all instructions include one or more of these fields. The MOD R/M byte 3602 includes a MOD field 3642, a register field 3644, and R/M field 3646.

[0203] The content of the MOD field 3642 distinguishes between memory access and non-memory access modes. In some embodiments, when the MOD field 3642 has a value of b11, a register-direct addressing mode is utilized, and otherwise register-indirect addressing is used.

[0204] The register field 3644 may encode either the destination register operand or a source register operand, or may encode an opcode extension and not be used to encode any instruction operand. The content of register index field 3644, directly or through address generation, specifies the locations of a source or destination operand (either in a register or in memory). In some embodiments, the register field 3644 is supplemented with an additional bit from a prefix (e.g., prefix 3501) to allow for greater addressing.

[0205] The R/M field 3646 may be used to encode an instruction operand that references a memory address, or may be used to encode either the destination register operand or a source register operand. Note the R/M field 3646 may be combined with the MOD field 3642 to dictate an addressing mode in some embodiments.

[0206] The SIB byte 3604 includes a scale field 3652, an index field 3654, and a base field 3656 to be used in the generation of an address. The scale field 3652 indicates scaling factor. The index field 3654 specifies an index register to use. In some embodiments, the index field 3654 is supplemented with an additional bit from a prefix (e.g., prefix 3501) to allow for greater addressing. The base field 3656 specifies a base register to use. In some embodiments, the base field 3656 is supplemented with an additional bit from a prefix (e.g., prefix 3501) to allow for greater addressing. In practice, the content of the scale field 3652 allows for the scaling of the content of the index field 3654 for memory address generation (e.g., for address generation that uses 2^{scale} * index + base).

[0207] Some addressing forms utilize a displacement value to generate a memory address. For example, a memory address may be generated according to 2^{scale} * index + base + displacement, index*scale+displacement, r/m + displacement, instruction pointer (RIP/EIP) + displacement, register + displacement, etc. The displacement may be a 1-byte, 2-byte, 4-byte, etc. value. In some embodiments, a displacement field 3507 provides this value. Additionally, in some embodiments, a displacement factor usage is encoded in the MOD field of the addressing field 3505 that indicates a compressed displacement scheme for which a displacement value is calculated by multiplying disp8 in conjunction with a scaling factor N that is determined based on the vector length, the value of a b bit, and the input element size of the instruction. The displacement value is stored in the displacement field 3507.

[0208] In some embodiments, an immediate field 3509 specifies an immediate for the instruction. An immediate may be encoded as a 1-byte value, a 2-byte value, a 4-byte value, etc.

[0209] **Figure 37** illustrates embodiments of a first prefix 3501(A). In some embodiments, the first prefix 3501(A) is an embodiment of a REX prefix. Instructions that use this prefix may specify general purpose registers, 64-bit packed data registers (e.g., single instruction, multiple data (SIMD) registers or vector registers), and/or control registers and debug registers (e.g., CR8-CR15 and DR8-DR15).

[0210] Instructions using the first prefix 3501(A) may specify up to three registers using 3-bit fields depending on the format: 1) using the reg field 3644 and the R/M field 3646 of the Mod R/M byte 3602; 2) using the Mod R/M byte 3602 with the SIB byte 3604 including using the reg field 3644 and the base field 3656 and index field 3654; or 3) using the register field of an opcode.

[0211] In the first prefix 3501(A), bit positions 7:4 are set as 0100. Bit position 3 (W) can be used to determine the operand size, but may not solely determine operand width. As such, when W = 0, the operand size is determined by a code segment descriptor (CS.D) and when W = 1, the operand size is 64-bit.

[0212] Note that the addition of another bit allows for 16 (2^4) registers to be addressed, whereas the MOD R/M reg field 3644 and MOD R/M R/M field 3646 alone can each only address 8 registers.

[0213] In the first prefix 3501(A), bit position 2 (R) may an extension of the MOD R/M reg field 3644 and may be used to modify the ModR/M reg field 3644 when that field encodes a general purpose register, a 64-bit packed data register (e.g., a SSE register), or a control or debug register. R is ignored when Mod R/M byte 3602 specifies other registers or defines an extended opcode.

[0214] Bit position 1 (X) X bit may modify the SIB byte index field 3654.

[0215] Bit position B (B) B may modify the base in the Mod R/M R/M field 3646 or the SIB byte base field 3656; or it may modify the opcode register field used for accessing general purpose registers (e.g., general purpose registers 3425).

[0216] **FIGS. 38(A)-(D)** illustrate embodiments of how the R, X, and B fields of the first prefix 3501(A) are used. **Figure 38(A)** illustrates R and B from the first prefix 3501(A) being used to extend the reg field 3644 and R/M field 3646 of the MOD R/M byte 3602 when the SIB byte 36 04 is not used for memory addressing. **Figure 38(B)** illustrates R and B from the first prefix 3501(A) being used to extend the reg field 3644 and R/M field 3646 of the MOD R/M byte 3602 when the SIB byte 36 04 is not used (register-register addressing). **Figure 38(C)** illustrates R, X, and B from the first prefix 3501(A) being used to extend the reg field 3644 of the MOD R/M byte 3602 and the index field 3654 and base field 3656 when the SIB byte 36 04 being used for memory addressing. **Figure 38(D)** illustrates B from the first prefix 3501(A) being used to extend the reg field 3644 of the MOD R/M byte 3602 when a register is encoded in the opcode 3503.

[0217] **FIGS. 39(A)-(B)** illustrate embodiments of a second prefix 3501(B). In some embodiments, the second prefix 3501(B) is an embodiment of a VEX prefix. The second prefix 3501(B) encoding allows instructions to have more than two operands, and allows SIMD vector registers (e.g., vector/SIMD registers 3410) to be longer than 64-bits (e.g., 128-bit and 256-bit). The use of the second prefix 3501(B) provides for three-operand (or more) syntax. For example, previous two-operand instructions performed operations such as A = A + B, which overwrites a source operand. The use of the second prefix 3501(B) enables operands to perform nondestructive operations such as A = B + C.

[0218] In some embodiments, the second prefix 3501(B) comes in two forms - a two-byte form and a three-byte form. The two-byte second prefix 3501(B) is used mainly for 128-bit, scalar, and some 256-bit instructions; while the three-byte second prefix 3501(B) provides a compact replacement of the first prefix 3501(A) and 3-byte opcode instructions.

[0219] **Figure 39(A)** illustrates embodiments of a two-byte form of the second prefix 3501(B). In one example, a format field 3901 (byte 0 3903) contains the value C5H. In one example, byte 1 3905 includes a "R" value in bit[7]. This value is the complement of the same value of the first prefix 3501(A). Bit[2] is used to dictate the length (L) of the vector (where a value of 0 is a scalar or 128-bit vector and a value of 1 is a 256-bit vector). Bits[1:0] provide opcode extensionality equivalent to some legacy prefixes (e.g., 00 = no prefix, 01 = 66H, 10 = F3H, and 11 = F2H). Bits[6:3] shown as vvvv may be used to: 1) encode the first source register operand, specified in inverted (1s complement) form and valid for instructions with 2 or more source operands; 2) encode the destination register operand, specified in 1s complement form for certain vector shifts; or 3) not encode any operand, the field is reserved and should contain a certain value, such as 1111b.

[0220] Instructions that use this prefix may use the Mod R/M R/M field 3646 to encode the instruction operand that references a memory address or encode either the destination register operand or a source register operand.

[0221] Instructions that use this prefix may use the Mod R/M reg field 3644 to encode either the destination register operand or a source register operand, be treated as an opcode extension and not used to encode any instruction operand.

[0222] For instruction syntax that support four operands, vvvv, the Mod R/M R/M field 3646 and the Mod R/M reg field 3644 encode three of the four operands. Bits[7:4] of the immediate 3509 are then used to encode the third source register operand.

[0223] **Figure 39(B)** illustrates embodiments of a three-byte form of the second prefix 3501(B). in one example, a format field 3911 (byte 0 3913) contains the value C4H. Byte 1 3915 includes in bits[7:5] "R," "X," and "B" which are the complements of the same values of the first prefix 3501(A). Bits[4:0] of byte 1 3915 (shown as mmmmm) include content to encode, as need, one or more implied leading opcode bytes. For example, 00001 implies a 0FH leading opcode, 00010 implies a 0F38H leading opcode, 00011 implies a leading 0F3AH opcode, etc.

[0224] Bit[7] of byte 2 3917 is used similar to W of the first prefix 3501(A) including helping to determine promotable operand sizes. Bit[2] is used to dictate the length (L) of the vector (where a value of 0 is a scalar or 128-bit vector and a value of 1 is a 256-bit vector). Bits[1:0] provide opcode extensionality equivalent to some legacy prefixes (e.g., 00 = no prefix, 01 = 66H, 10 = F3H, and 11 = F2H). Bits[6:3], shown as vvvv, may be used to: 1) encode the first source register operand, specified in inverted (1s complement) form and valid for instructions with 2 or more source operands; 2) encode the destination register operand, specified in 1s complement form for certain vector shifts; or 3) not encode any operand, the field is reserved and should contain a certain value, such as 1111b.

[0225] Instructions that use this prefix may use the Mod R/M R/M field 3646 to encode the instruction operand that references a memory address or encode either the destination register operand or a source register operand.

[0226] Instructions that use this prefix may use the Mod R/M reg field 3644 to encode either the destination register operand or a source register operand, be treated as an opcode extension and not used to encode any instruction operand.

[0227] For instruction syntax that support four operands, vvvv, the Mod R/M R/M field 3646, and the Mod R/M reg field 3644 encode three of the four operands. Bits[7:4] of the immediate 3509 are then used to encode the third source register operand.

[0228] **Figure 40** illustrates embodiments of a third prefix 3501(C). In some embodiments, the first prefix 3501(A) is an embodiment of an EVEX prefix. The third prefix 3501(C) is a four-byte prefix.

[0229] The third prefix 3501(C) can encode 32 vector registers (e.g., 128-bit, 256-bit, and 512-bit registers) in 64-bit mode. In some embodiments, instructions that utilize a writemask/opmask (see discussion of registers in a previous figure, such as **Figure 34)** or predication utilize this prefix. Opmask register allow for conditional processing or selection control. Opmask instructions, whose source/destination operands are opmask registers and treat the content of an opmask register as a single value, are encoded using the second prefix 3501(B).

[0230] The third prefix 3501(C) may encode functionality that is specific to instruction classes (e.g., a packed instruction with "load+op" semantic can support embedded broadcast functionality, a floating-point instruction with rounding semantic can support static rounding functionality, a floating-point instruction with non-rounding arithmetic semantic can support "suppress all exceptions" functionality, etc.).

[0231] The first byte of the third prefix 3501(C) is a format field 4011 that has a value, in one example, of 62H. Subsequent bytes are referred to as payload bytes 4015-4019 and collectively form a 24-bit value of P[23:0] providing specific capability in the form of one or more fields (detailed herein).

[0232] In some embodiments, P[1:0] of payload byte 4019 are identical to the low two mmmmm bits. P[3:2] are reserved in some embodiments. Bit P[4] (R') allows access to the high 16 vector register set when combined with P[7] and the ModR/M reg field 3644. P[6] can also provide access to a high 16 vector register when SIB-type addressing is not needed. P[7:5] consist of an R, X, and B which are operand specifier modifier bits for vector register, general purpose register, memory addressing and allow access to the next set of 8 registers beyond the low 8 registers when combined with the ModR/M register field 3644 and ModR/M R/M field 3646. P[9:8] provide opcode extensionality equivalent to some legacy prefixes (e.g., 00 = no prefix, 01 = 66H, 10 = F3H, and 11 = F2H). P[10] in some embodiments is a fixed value of 1. P[14:11], shown as vvvv, may be used to: 1) encode the first source register operand, specified in inverted (1s complement) form and valid for instructions with 2 or more source operands; 2) encode the destination register operand, specified in 1s complement form for certain vector shifts; or 3) not encode any operand, the field is reserved and should contain a certain value, such as 1111b.

[0233] P[15] is similar to W of the first prefix 3501(A) and second prefix 3511(B) and may serve as an opcode extension bit or operand size promotion.

[0234] P[18:16] specify the index of a register in the opmask (writemask) registers (e.g., writemask/predicate registers 3415). In one embodiment of the invention, the specific value aaa = 000 has a special behavior implying no opmask is used for the particular instruction (this may be implemented in a variety of ways including the use of a opmask hardwired to all ones or hardware that bypasses the masking hardware). When merging, vector masks allow any set of elements in the destination to be protected from updates during the execution of any operation (specified by the base operation and the augmentation operation); in other one embodiment, preserving the old value of each element of the destination where the corresponding mask bit has a 0. In contrast, when zeroing vector masks allow any set of elements in the destination to be zeroed during the execution of any operation (specified by the base operation and the augmentation operation); in one embodiment, an element of the destination is set to 0 when the corresponding mask bit has a 0 value. A subset of this functionality is the ability to control the vector length of the operation being performed (that is, the span of elements being modified, from the first to the last one); however, it is not necessary that the elements that are modified be consecutive. Thus, the opmask field allows for partial vector operations, including loads, stores, arithmetic, logical, etc. While embodiments of the invention are described in which the opmask field's content selects one of a number of opmask registers that contains the opmask to be used (and thus the opmask field's content indirectly identifies that masking to be performed), alternative embodiments instead or additional allow the mask write field's content to directly specify the masking to be performed.

[0235] P[19] can be combined with P[14:11] to encode a second source vector register in a non-destructive source syntax which can access an upper 16 vector registers using P[19]. P[20] encodes multiple functionalities, which differs across different classes of instructions and can affect the meaning of the vector length/ rounding control specifier field (P[22:21]). P[23] indicates support for merging-writemasking (e.g., when set to 0) or support for zeroing and merging-writemasking (e.g., when set to 1).

[0236] Exemplary embodiments of encoding of registers in instructions using the third prefix 3501(C) are detailed in the following tables.

Table 1: 32-Register Support in 64-bit Mode

	4	**3**	**[2:0]**	**REG. TYPE**	**COMMON USAGES**
REG	R'	R	ModR/M reg	GPR, Vector	Destination or Source
vvvv	V'	vvvv		GPR, Vector	2nd Source or Destination
RM	X	B	ModR/M R/M	GPR, Vector	1st Source or Destination
BASE	0	B	ModR/M R/M	GPR	Memory addressing
INDEX	0	X	SIB.index	GPR	Memory addressing
VIDX	V'	X	SIB.index	Vector	VSIB memory addressing

Table 2: Encoding Register Specifiers in 32-bit Mode

	[2:0]	**REG. TYPE**	**COMMON USAGES**
REG	ModR/M reg	GPR, Vector	Destination or Source
vvvv	vvvv	GPR, Vector	2nd Source or Destination
RM	ModR/M R/M	GPR, Vector	1st Source or Destination
BASE	ModR/M R/M	GPR	Memory addressing
INDEX	SIB.index	GPR	Memory addressing
VIDX	SIB.index	Vector	VSIB memory addressing

Table 3: Opmask Register Specifier Encoding

	[2:0]	**REG. TYPE**	**COMMON USAGES**
REG	ModR/M Reg	k0-k7	Source

(continued)

	[2:0]	REG. TYPE	COMMON USAGES
vvvv	vvvv	k0-k7	2nd Source
RM	ModR/M R/M	k0-7	1st Source
{k1]	aaa	k0^1-k7	Opmask

[0237] Program code may be applied to input instructions to perform the functions described herein and generate output information. The output information may be applied to one or more output devices, in known fashion. For purposes of this application, a processing system includes any system that has a processor, such as, for example, a digital signal processor (DSP), a microcontroller, an application specific integrated circuit (ASIC), or a microprocessor.

[0238] The program code may be implemented in a high-level procedural or object-oriented programming language to communicate with a processing system. The program code may also be implemented in assembly or machine language, if desired. In fact, the mechanisms described herein are not limited in scope to any particular programming language. In any case, the language may be a compiled or interpreted language.

[0239] Embodiments of the mechanisms disclosed herein may be implemented in hardware, software, firmware, or a combination of such implementation approaches. Embodiments of the invention may be implemented as computer programs or program code executing on programmable systems comprising at least one processor, a storage system (including volatile and non-volatile memory and/or storage elements), at least one input device, and at least one output device.

[0240] One or more aspects of at least one embodiment may be implemented by representative instructions stored on a machine-readable medium which represents various logic within the processor, which when read by a machine causes the machine to fabricate logic to perform the techniques described herein. Such representations, known as "IP cores" may be stored on a tangible, machine readable medium and supplied to various customers or manufacturing facilities to load into the fabrication machines that actually make the logic or processor.

[0241] Such machine-readable storage media may include, without limitation, non-transitory, tangible arrangements of articles manufactured or formed by a machine or device, including storage media such as hard disks, any other type of disk including floppy disks, optical disks, compact disk read-only memories (CD-ROMs), compact disk rewritable's (CD-RWs), and magneto-optical disks, semiconductor devices such as read-only memories (ROMs), random access memories (RAMs) such as dynamic random access memories (DRAMs), static random access memories (SRAMs), erasable programmable read-only memories (EPROMs), flash memories, electrically erasable programmable read-only memories (EEPROMs), phase change memory (PCM), magnetic or optical cards, or any other type of media suitable for storing electronic instructions.

[0242] Accordingly, embodiments of the invention also include non-transitory, tangible machine-readable media containing instructions or containing design data, such as Hardware Description Language (HDL), which defines structures, circuits, apparatuses, processors and/or system features described herein. Such embodiments may also be referred to as program products.

Emulation (including binary translation, code morphing, etc.)

[0243] In some cases, an instruction converter may be used to convert an instruction from a source instruction set to a target instruction set. For example, the instruction converter may translate (e.g., using static binary translation, dynamic binary translation including dynamic compilation), morph, emulate, or otherwise convert an instruction to one or more other instructions to be processed by the core. The instruction converter may be implemented in software, hardware, firmware, or a combination thereof. The instruction converter may be on processor, off processor, or part on and part off processor.

[0244] **Figure 41** illustrates a block diagram contrasting the use of a software instruction converter to convert binary instructions in a source instruction set to binary instructions in a target instruction set according to embodiments of the invention. In the illustrated embodiment, the instruction converter is a software instruction converter, although alternatively the instruction converter may be implemented in software, firmware, hardware, or various combinations thereof. **Figure 41** shows a program in a high level language 4102 may be compiled using a first ISA compiler 4104 to generate first ISA binary code 4106 that may be natively executed by a processor with at least one first instruction set core 4116. The processor with at least one first ISA instruction set core 4116 represents any processor that can perform substantially the same functions as an Intel® processor with at least one first ISA instruction set core by compatibly executing or otherwise processing (1) a substantial portion of the instruction set of the first ISA instruction set core or (2) object code versions of applications or other software targeted to run on an Intel processor with at least one first ISA instruction set core, in order to achieve substantially the same result as a processor with at least one first ISA instruction set core. The first ISA compiler 4104 represents a compiler that is operable to generate first ISA binary code 4106 (e.g., object code) that can, with or without additional linkage processing, be executed on the processor with at least one first ISA instruction set core 4116. Similarly, **Figure 41** shows the program in the high level language 4102 may be compiled using an alternative instruction set compiler 4108 to generate alternative instruction set binary code 4110 that may be natively executed by a processor without a first ISA instruction set core 4114. The instruction converter 4112 is used to convert the first ISA binary code 4106 into code that may be natively executed by the processor without a first ISA instruction set core 4114. This converted code is not likely to be the same as the alternative instruction set binary code 4110 because an instruction converter capable of this is difficult to make; however, the converted code will accomplish the general operation and be made up of instructions from the alternative instruction set. Thus, the instruction converter 4112 represents software, firmware, hardware, or a combination thereof that, through emulation, simulation or any other process, allows a processor or other electronic device that does not have a first ISA instruction set processor or core to execute the first ISA binary code 4106.

[0245] References to "one embodiment," "an embodiment," "an example embodiment," etc., indicate that the embodiment described may include a particular feature, structure, or characteristic, but every embodiment may not necessarily include the particular feature, structure, or characteristic. Moreover, such phrases are not necessarily referring to the same embodiment. Further, when a particular feature, structure, or characteristic is described in connection with an embodiment, it is submitted that it is within the knowledge of one skilled in the art to affect such feature, structure, or characteristic in connection with other embodiments whether or not explicitly described.

[0246] Moreover, in the various embodiments described above, unless specifically noted otherwise, disjunctive language such as the phrase "at least one of A, B, or C" is intended to be understood to mean either A, B, or C, or any combination thereof (e.g., A, B, and/or C). As such, disjunctive language is not intended to, nor should it be understood to, imply that a given embodiment requires at least one of A, at least one of B, or at least one of C to each be present.

[0247] Exemplary embodiments include, but are not limited to:

1. An apparatus comprising:

    decode circuitry to decode a single instruction having fields for an opcode, an indication of a location of a first source operand, an indication of a location of a second source operand, and an indication of a location of a destination operand, wherein the opcode is to indicate that execution circuitry is to at least convert data elements of the first and second source operands from a first floating point representation to a second floating point representation, perform matrix multiplication with the converted data elements, and accumulate results of the matrix multiplication in the destination operand in the first floating point representation; and
    the execution circuitry to execute to the decoded instruction as specified by the opcode.

2. The apparatus of example 1, wherein the execution circuitry is further to execute the decoded instruction to convert the data elements from the first and second source operands from a signaling not-a-number value to a quiet non-a-number value.

3. The apparatus of any of examples 1-2, wherein the first floating point representation is single precision floating point.

4. The apparatus of any of examples 1-3, wherein the second floating point representation is a 19-bit floating point number.

5. The apparatus of example 4, wherein the 19-bit floating-point number representation has a sign bit, 8 exponent bits, and 10 mantissa bits that are the corresponding most significant bits of a single precision floating number.

6. The apparatus of any of examples 1-5, wherein the execution circuitry is further to execute the decoded instruction to zero any remaining data elements of the destination operand.

7. A non-transitory machine readable medium storing at least an instance of a single instruction, wherein a processor is to respond to the single instruction to perform a method comprising:

    decoding the single instruction having fields for an opcode, an indication of a location of a first source operand, an indication of a location of a second source operand, and an indication of a location of a destination operand, wherein the opcode is to indicate that execution circuitry is to at least convert data elements of the first and second source operands from a first floating point representation to a second floating point representation, perform matrix multiplication with the converted data elements, and accumulate results of the matrix multiplication

in the destination operand in the first floating point representation;
executing the decoded instruction to perform operations as specified by the opcode.

8. The non-transitory machine readable medium of example 7, wherein the executing further comprises converting the data elements from the first and second source operands from a signaling not-a-number value to a quiet non-a-number value.

9. The non-transitory machine readable medium of any of examples 7-8, wherein the first floating point representation is single precision floating point.

10. The non-transitory machine readable medium of any of examples 7-9, wherein the second floating point representation is a 19-bit floating-point number.

11. The non-transitory machine readable medium of example 10, wherein the 19-bit floating-point number representation has a lower 13 bits set to zero.

12. The non-transitory machine readable medium of any of examples 7-11, wherein the executing further comprises zeroing any remaining data elements of the destination operand.

13. The non-transitory machine readable medium of any of examples 7-12, further comprising:
translating the single instruction into one or more instructions of a different instruction set, wherein the executing the decoded instruction as specified by the opcode comprises executing the one or more instructions of the different instruction set to perform the operations as specified by the opcode of the single instruction.

14. An apparatus comprising:

decode circuitry to decode a single instruction having fields for an opcode, an indication of a location of a first source operand, an indication of a location of a second source operand, and an indication of a location of a destination operand, wherein the opcode is to indicate that execution circuitry is to at least convert data elements of the first and second source operands from a first floating point representation to a second floating point representation, perform matrix multiplication with a transpose of the converted data elements of the first source operand and the non-transposed, converted data elements of the second source operand, and accumulate results of the matrix multiplication in the destination operand in the first floating point representation; and
the execution circuitry to execute to the decoded instruction as specified by the opcode.

15. The apparatus of example 14, wherein the execution circuitry is further to execute the decoded instruction to convert the data elements from the first and second source operands from a signaling not-a-number value to a quiet non-a-number value.

16. The apparatus of any of examples 14-15, wherein the first floating point representation is single precision floating point.

17. The apparatus of examples 14-16, wherein the second floating point representation is a 19-bit floating-point number.

18. The apparatus of examples 14-17, wherein the 19-bit floating-point number representation has a lower 13 bits set to zero.

19. The apparatus of examples 14-18, wherein the execution circuitry is further to execute the decoded instruction to zero any remaining data elements of the destination operand.

20. A non-transitory machine-readable medium storing at least an instance of a single instruction, wherein a processor is to respond to the single instruction to perform a method comprising:

decoding the single instruction having fields for an opcode, an indication of a location of a first source operand, an indication of a location of a second source operand, and an indication of a location of a destination operand, wherein the opcode is to indicate that execution circuitry is to at least convert data elements of the first and second source operands from a first floating point representation to a second floating point representation,

perform matrix multiplication with a transpose of the converted data elements of the first source operand and the non-transposed, converted data elements of the second source operand, and accumulate results of the matrix multiplication in the destination operand in the first floating point representation;
executing the decoded instruction as specified by the opcode.

[0248] The specification and drawings are, accordingly, to be regarded in an illustrative rather than a restrictive sense. It will, however, be evident that various modifications and changes may be made thereunto without departing from the broader spirit and scope of the disclosure as set forth in the claims.

Claims

1. An apparatus comprising:

   decode circuitry to decode a single instruction having fields for an opcode, an indication of a location of a first source operand, an indication of a location of a second source operand, and an indication of a location of a destination operand, wherein the opcode is to indicate that execution circuitry is to at least convert data elements of the first and second source operands from a first floating point representation to a second floating point representation, perform matrix multiplication with the converted data elements, and accumulate results of the matrix multiplication in the destination operand in the first floating point representation; and
   the execution circuitry to execute to the decoded instruction as specified by the opcode.

2. The apparatus of claim 1, wherein the execution circuitry is further to execute the decoded instruction to convert the data elements from the first and second source operands from a signaling not-a-number value to a quiet non-a-number value.

3. The apparatus of any of claims 1-2, wherein the first floating point representation is single precision floating point.

4. The apparatus of claim 3, wherein the second floating point representation is a 19-bit floating point number.

5. The apparatus of claim 4, wherein the 19-bit floating-point number representation has a sign bit, 8 exponent bits, and 10 mantissa bits that are the corresponding most significant bits of a single precision floating number.

6. The apparatus of any of claims 1-5, wherein the execution circuitry is further to execute the decoded instruction to zero any remaining data elements of the destination operand.

7. A non-transitory machine readable medium storing at least an instance of a single instruction, wherein a processor is to respond to the single instruction to perform a method comprising:

   decoding the single instruction having fields for an opcode, an indication of a location of a first source operand, an indication of a location of a second source operand, and an indication of a location of a destination operand, wherein the opcode is to indicate that execution circuitry is to at least convert data elements of the first and second source operands from a first floating point representation to a second floating point representation, perform matrix multiplication with the converted data elements, and accumulate results of the matrix multiplication in the destination operand in the first floating point representation;
   executing the decoded instruction to perform operations as specified by the opcode.

8. The non-transitory machine readable medium of claim 7, wherein the executing further comprises converting the data elements from the first and second source operands from a signaling not-a-number value to a quiet non-a-number value.

9. The non-transitory machine readable medium of any of claims 7-8, wherein the first floating point representation is single precision floating point.

10. The non-transitory machine readable medium of claim 9, wherein the second floating point representation is a 19-bit floating-point number.

11. The non-transitory machine readable medium of claim 10, wherein the 19-bit floating-point number representation

has a lower 13 bits set to zero.

12. The non-transitory machine readable medium of claim 9, wherein the executing further comprises zeroing any remaining data elements of the destination operand.

13. The non-transitory machine readable medium of claim 9, further comprising:
translating the single instruction into one or more instructions of a different instruction set, wherein the executing the decoded instruction as specified by the opcode comprises executing the one or more instructions of the different instruction set to perform the operations as specified by the opcode of the single instruction.

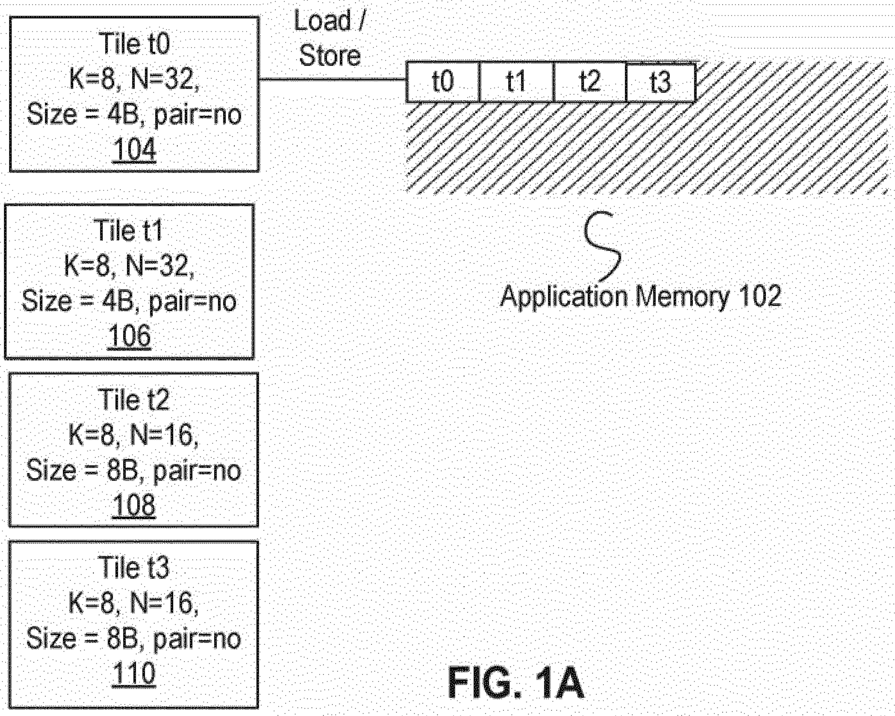

Tile t0
K=8, N=32,
Size = 4B, pair=no
104
Load / Store
t0
t1
t2
t3
Tile t1
K=8, N=32,
Size = 4B, pair=no
106
Application Memory 102
Tile t2
K=8, N=16,
Size = 8B, pair=no
108
Tile t3
K=8, N=16,
Size = 8B, pair=no
110

FIG. 1A

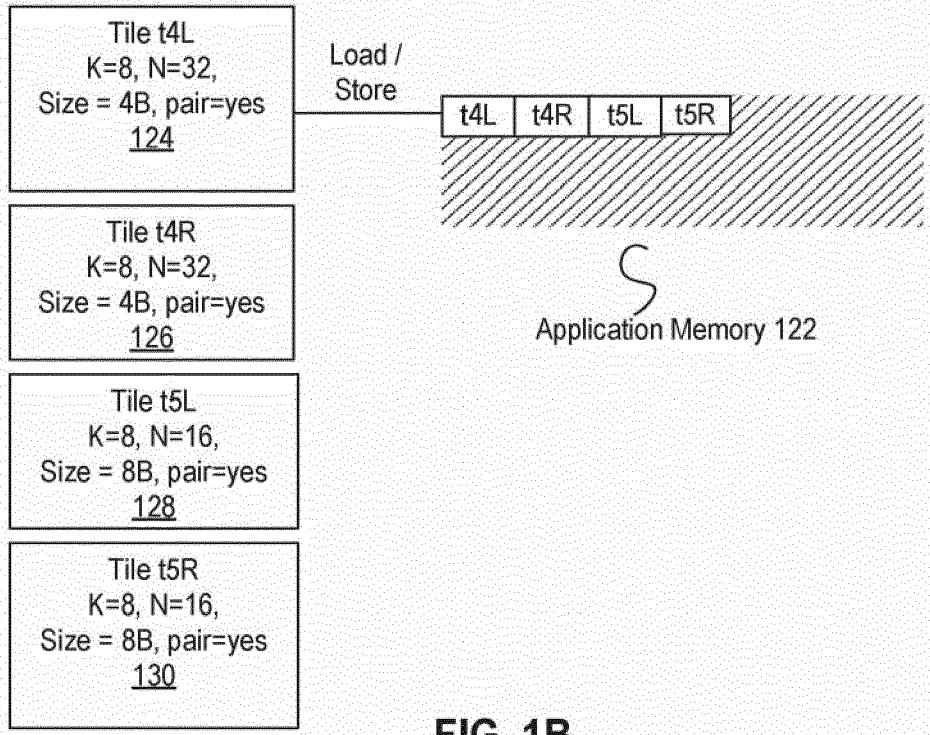

Tile t4L
K=8, N=32,
Size = 4B, pair=yes
124
Load / Store
t4L
t4R
t5L
t5R
Application Memory 122
Tile t4R
K=8, N=32,
Size = 4B, pair=yes
126
Tile t5L
K=8, N=16,
Size = 8B, pair=yes
128
Tile t5R
K=8, N=16,
Size = 8B, pair=yes
130

FIG. 1B

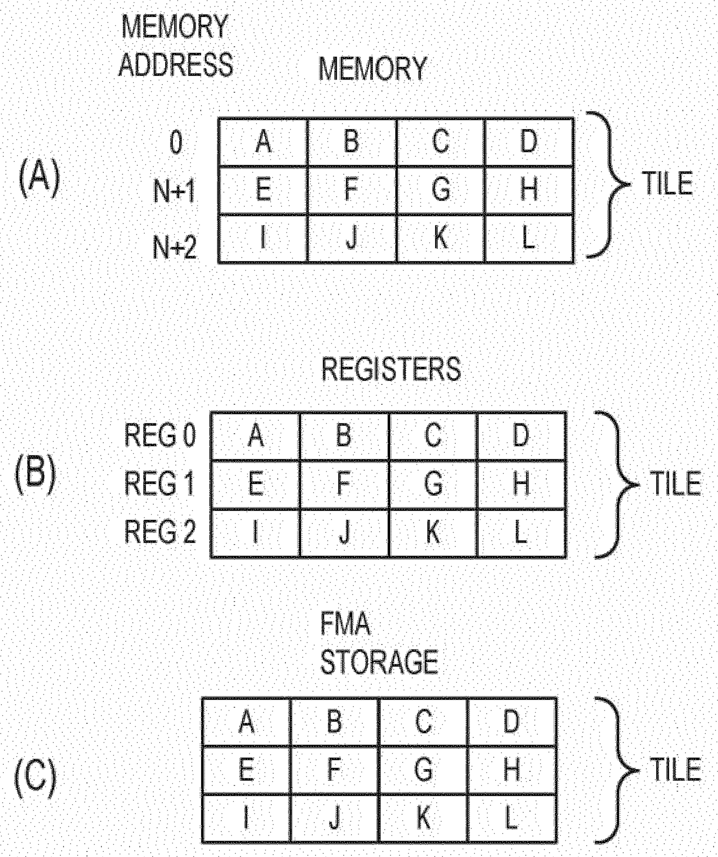

(A)
MEMORY ADDRESS
MEMORY
0
N+1
N+2
A
B
C
D
E
F
G
H
I
J
K
L
TILE
(B)
REGISTERS
REG 0
REG 1
REG 2
A
B
C
D
E
F
G
H
I
J
K
L
TILE
(C)
FMA STORAGE
A
B
C
D
E
F
G
H
I
J
K
L
TILE

FIG. 2

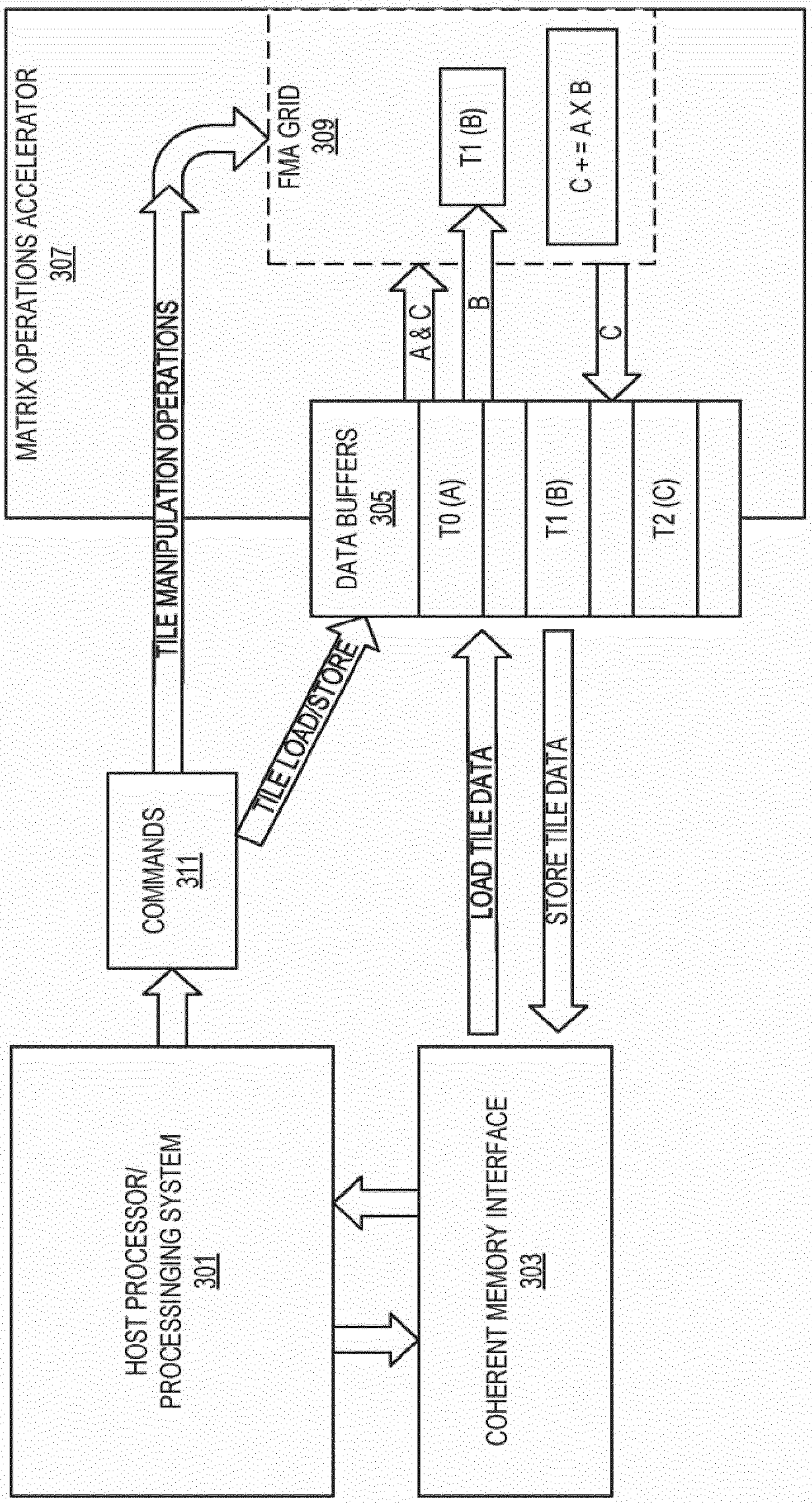

MATRIX OPERATIONS ACCELERATOR
307
FMA GRID
309
T1 (B)
C += A X B
TILE MANIPULATION OPERATIONS
A & C
B
C
DATA BUFFERS
305
T0 (A)
T1 (B)
T2 (C)
COMMANDS
311
TILE LOAD/STORE
LOAD TILE DATA
STORE TILE DATA
HOST PROCESSOR/
PROCESSINGING SYSTEM
301
COHERENT MEMORY INTERFACE
303

FIG. 3

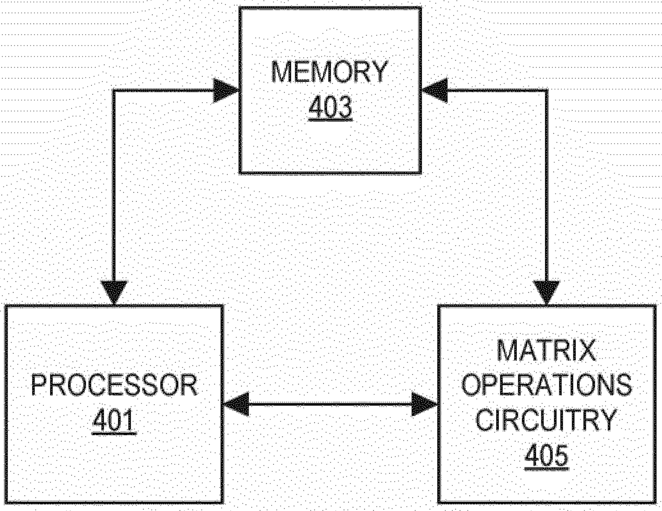

MEMORY
403
PROCESSOR
401
MATRIX OPERATIONS CIRCUITRY
405

FIG. 4

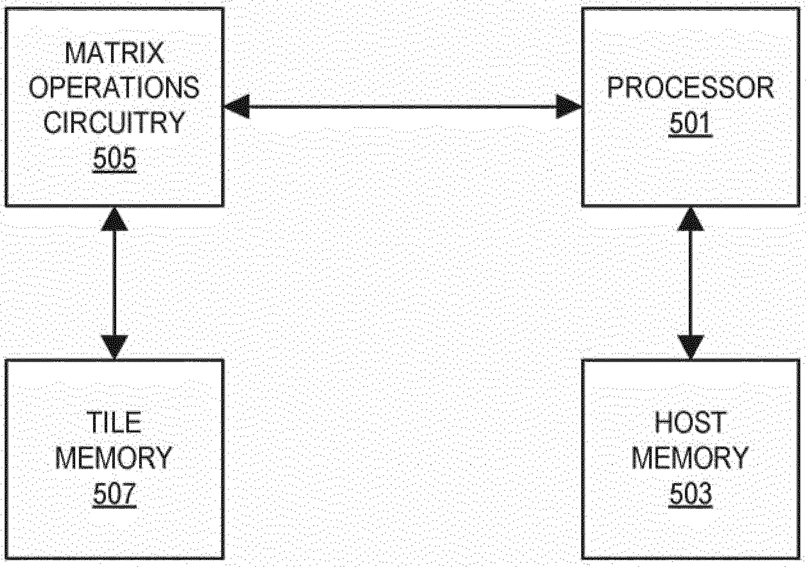

MATRIX OPERATIONS CIRCUITRY
505
PROCESSOR
501
TILE MEMORY
507
HOST MEMORY
503

FIG. 5

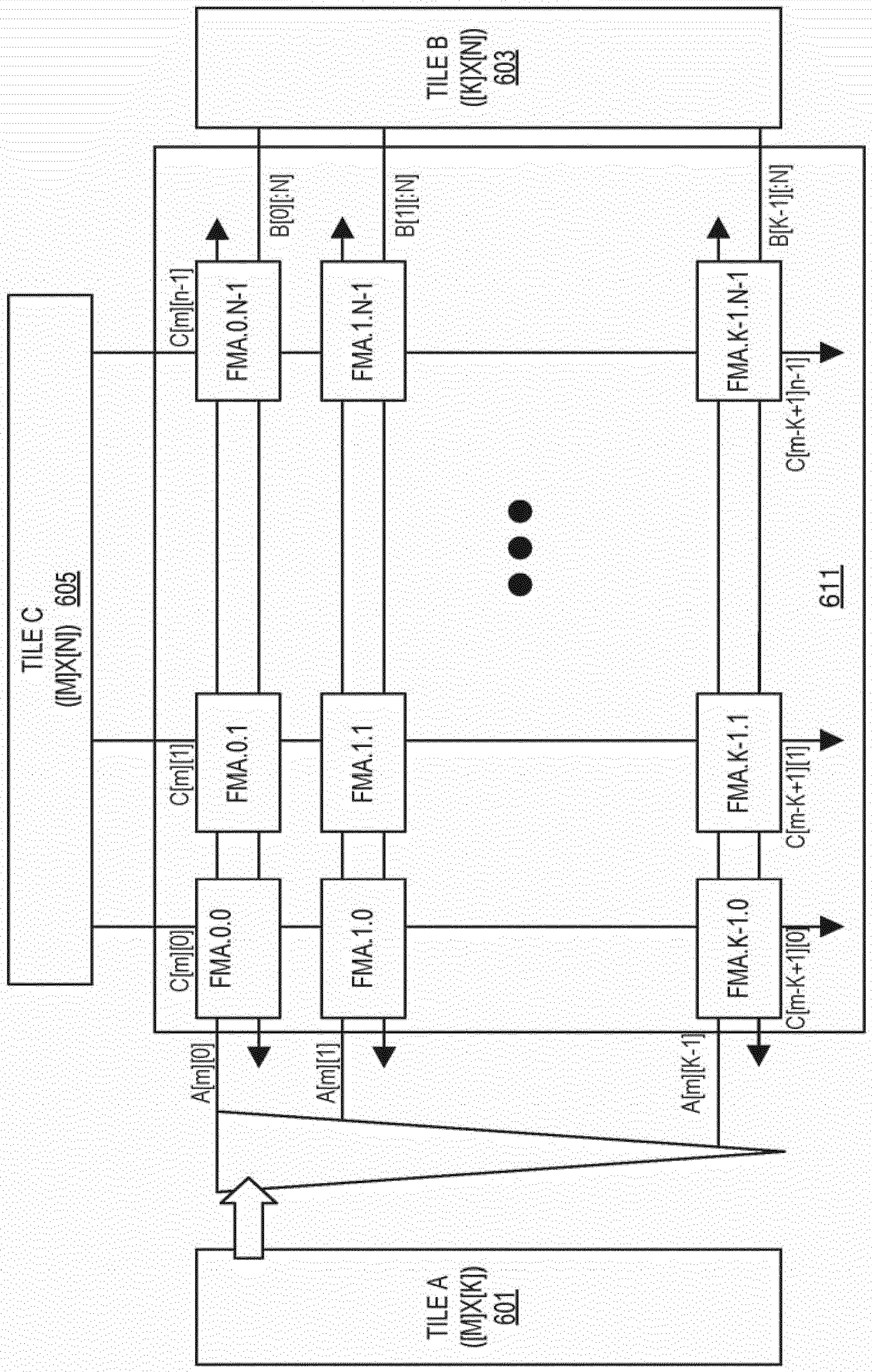

TILE A
([M]X[K])
601
TILE B
([K]X[N])
603
TILE C
([M]X[N]) 605
611
A[m][0]
A[m][1]
A[m][K-1]
B[0][:N]
B[1][:N]
B[K-1][:N]
C[m][0]
C[m][1]
C[m][n-1]
C[m-K+1][0]
C[m-K+1][1]
C[m-K+1][n-1]
FMA.0.0
FMA.0.1
FMA.0.N-1
FMA.1.0
FMA.1.1
FMA.1.N-1
FMA.K-1.0
FMA.K-1.1
FMA.K-1.N-1

FIG. 6

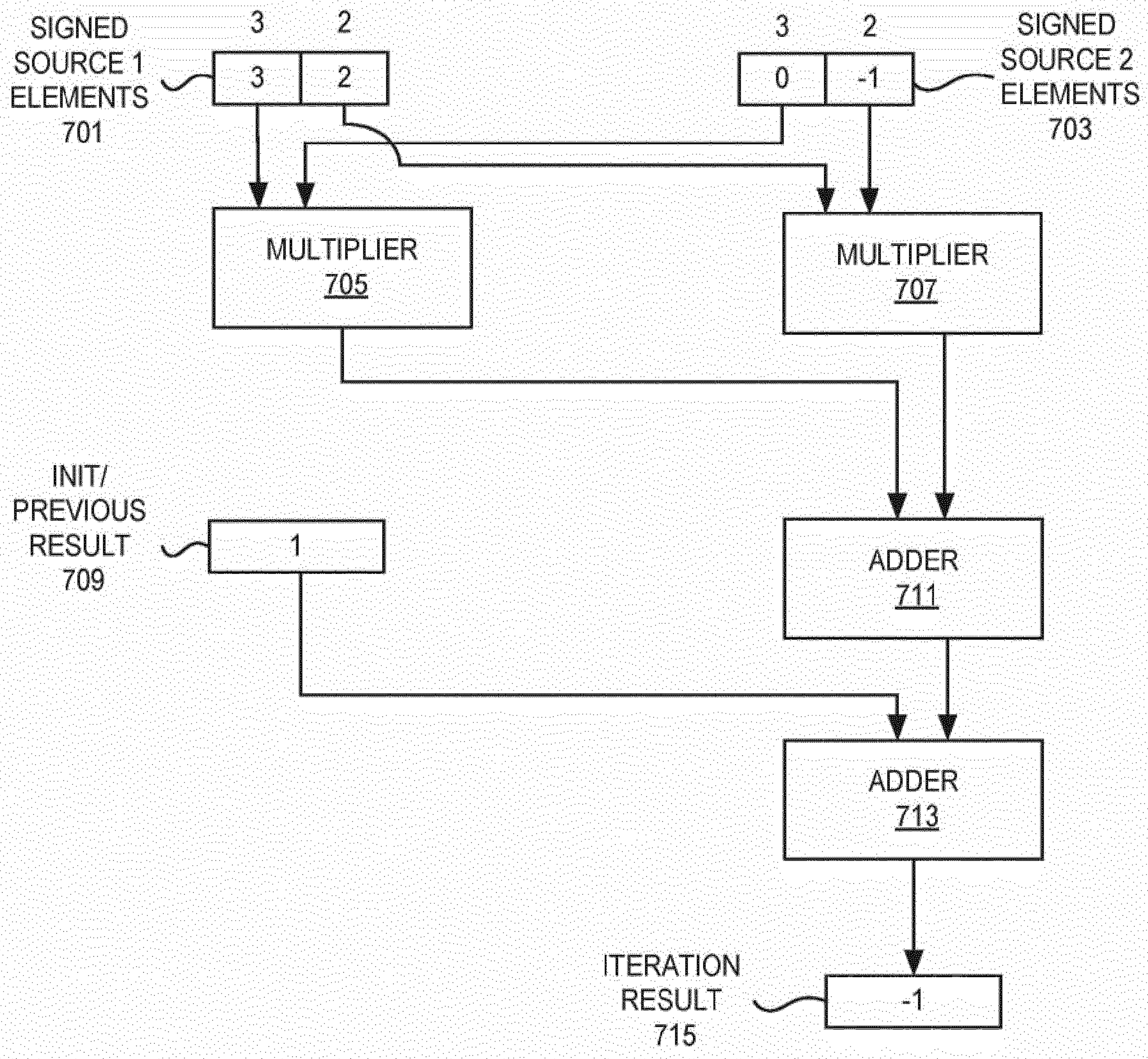

SIGNED SOURCE 1 ELEMENTS 701
3
2
3
2
3
2
0
-1
SIGNED SOURCE 2 ELEMENTS 703
MULTIPLIER 705
MULTIPLIER 707
INIT/ PREVIOUS RESULT 709
1
ADDER 711
ADDER 713
ITERATION RESULT 715
-1

FIG. 7

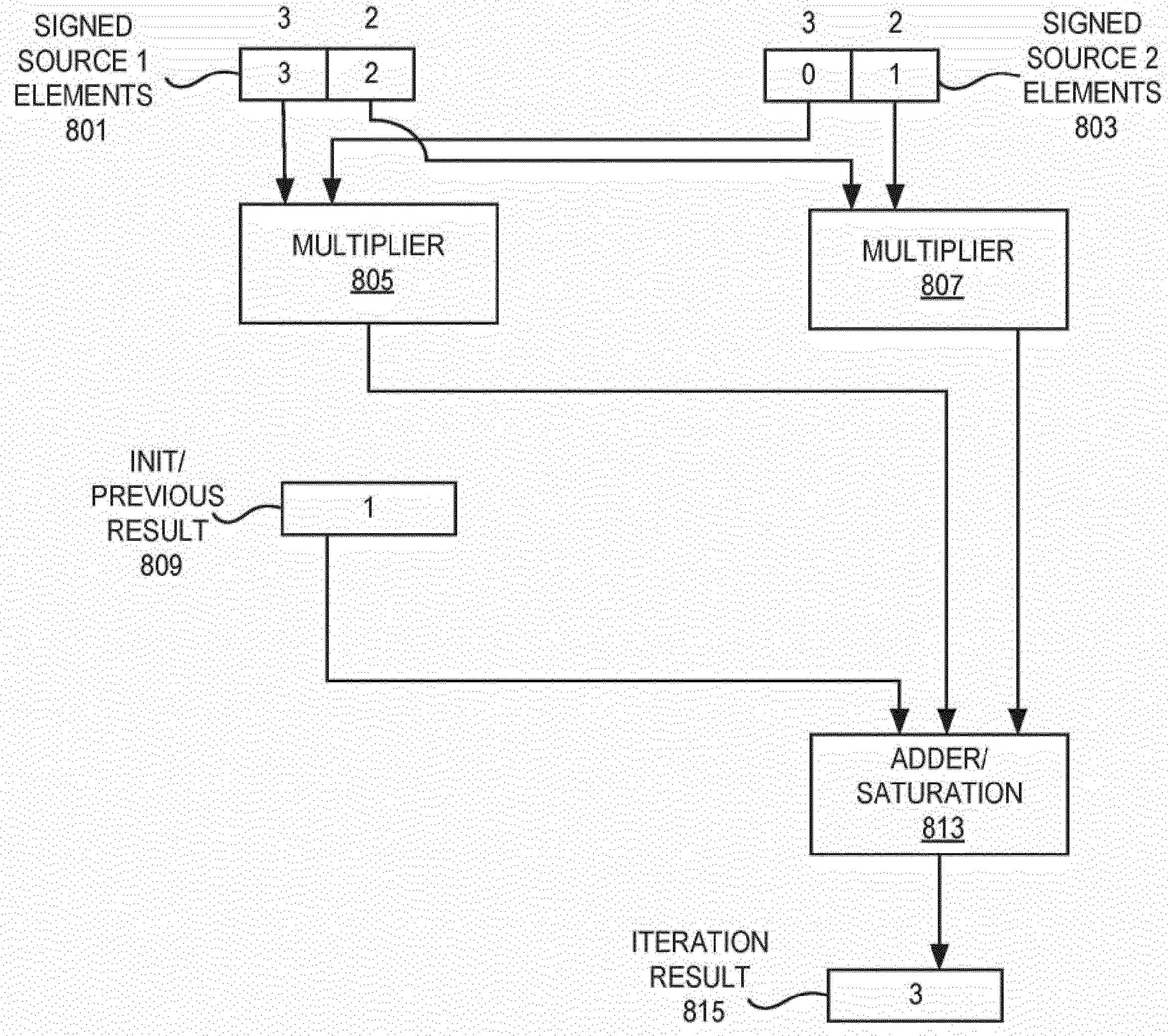

SIGNED SOURCE 1 ELEMENTS 801
3
2
3
2
SIGNED SOURCE 2 ELEMENTS 803
3
2
0
1
MULTIPLIER 805
MULTIPLIER 807
INIT/ PREVIOUS RESULT 809
1
ADDER/ SATURATION 813
ITERATION RESULT 815
3

FIG. 8

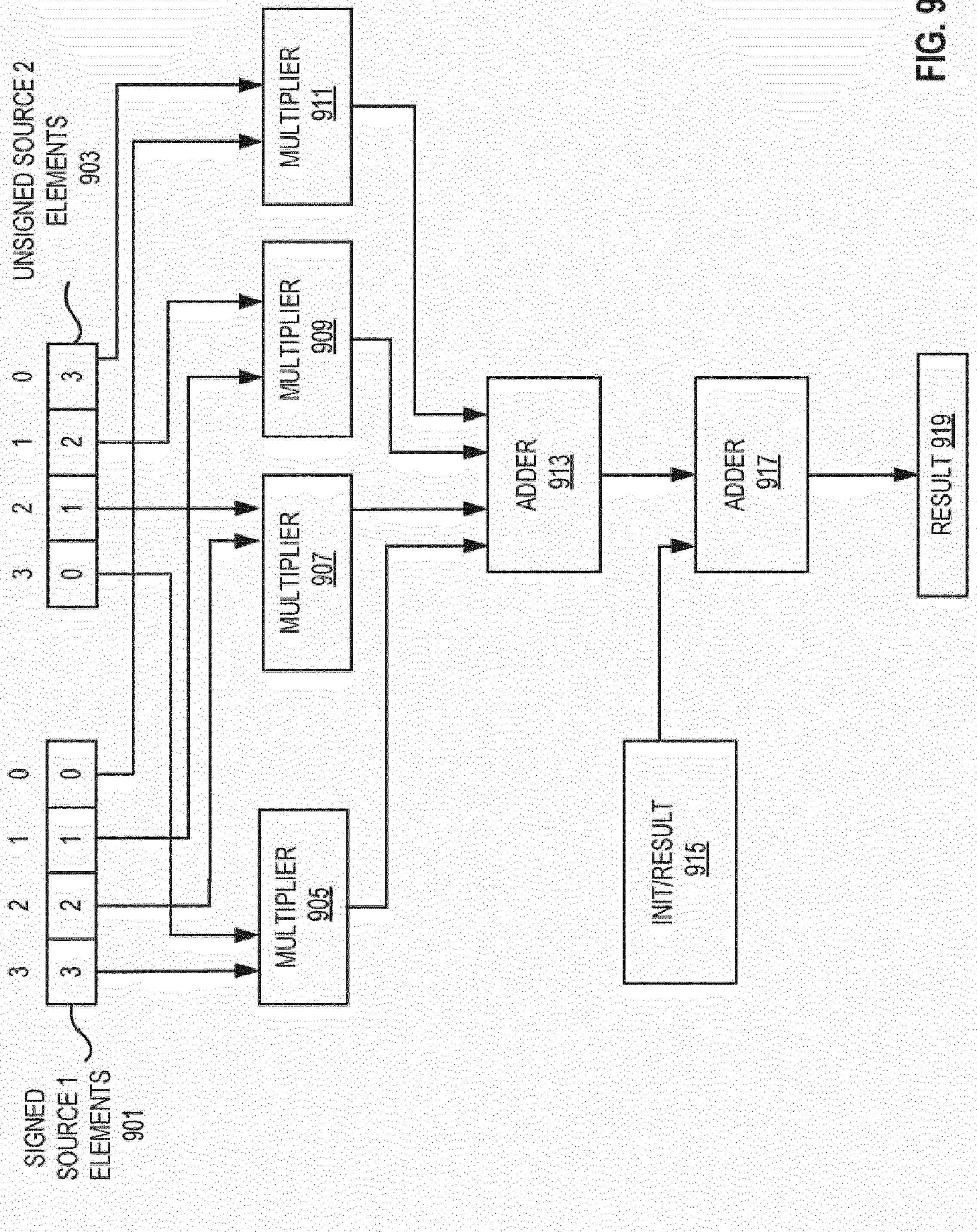

SIGNED SOURCE 1 ELEMENTS 901
3 2 1 0
3 2 1 0
UNSIGNED SOURCE 2 ELEMENTS 903
3 2 1 0
0 1 2 3
MULTIPLIER 905
MULTIPLIER 907
MULTIPLIER 909
MULTIPLIER 911
ADDER 913
INIT/RESULT 915
ADDER 917
RESULT 919

FIG. 9

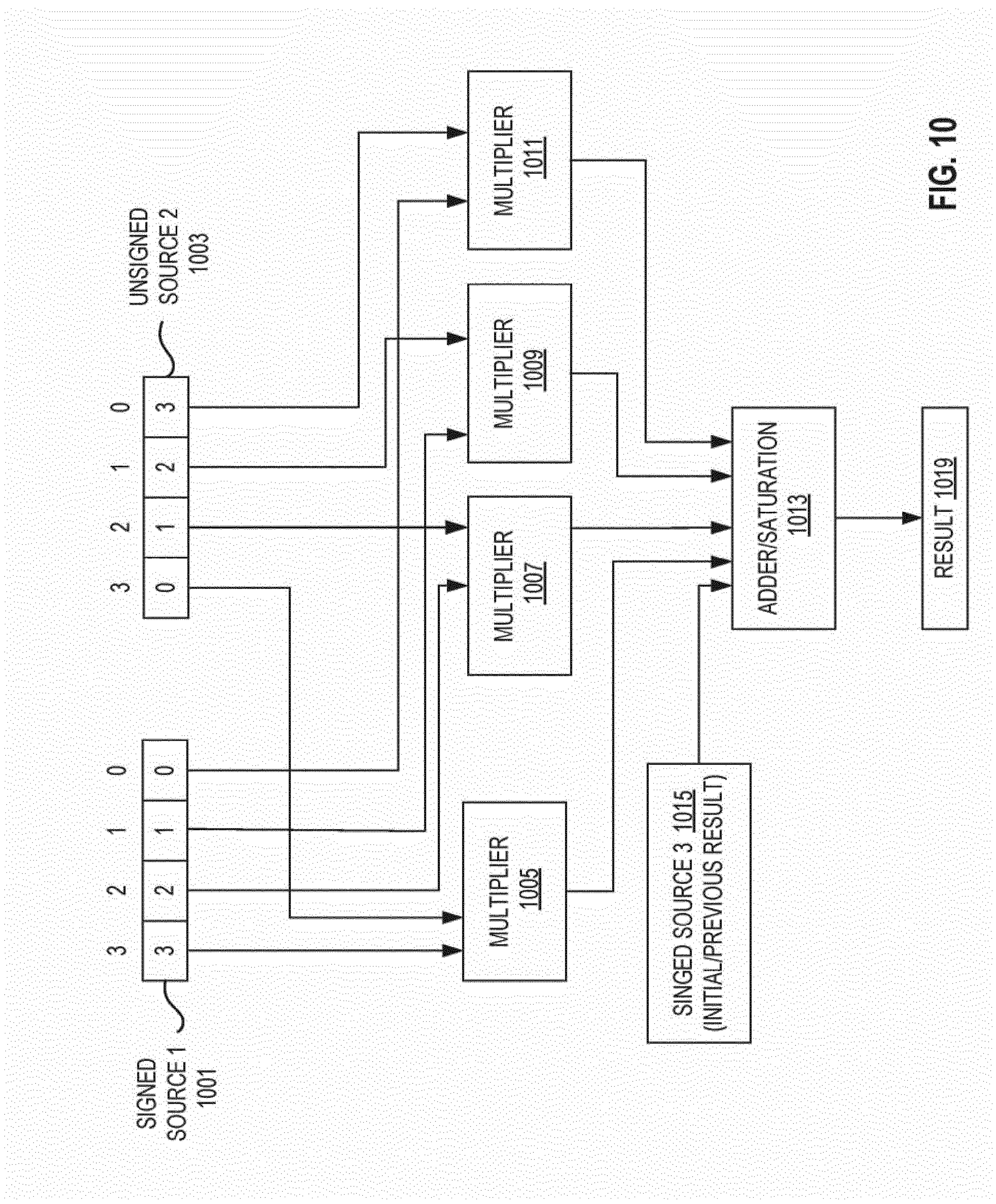

SIGNED SOURCE 1 1001
3 2 1 0
3 2 1 0
UNSIGNED SOURCE 2 1003
3 2 1 0
0 1 2 3
MULTIPLIER 1005
MULTIPLIER 1007
MULTIPLIER 1009
MULTIPLIER 1011
SINGED SOURCE 3 1015 (INITIAL/PREVIOUS RESULT)
ADDER/SATURATION 1013
RESULT 1019

FIG. 10

ACCUMULATOR 2X INPUT SIZES 1101

SOURCES	BITS	ACCUMULATOR	BITS
BYTE	8	WORD/HPFP	16
WORD	16	INT32/SPFP	32
SPFP/INT32	32	INT64/DPFP	64

ACCUMULATOR 4X INPUT SIZES 1103

SOURCES	BITS	ACCUMULATOR	BITS
BYTE	8	INT32/SPFP	32
WORD	16	INT64/DPFP	64

ACCUMULATOR 8X INPUT SIZES 1105

SOURCES	BITS	ACCUMULATOR	BITS
BYTE	8	INT64/DPFP	64

FIG. 11

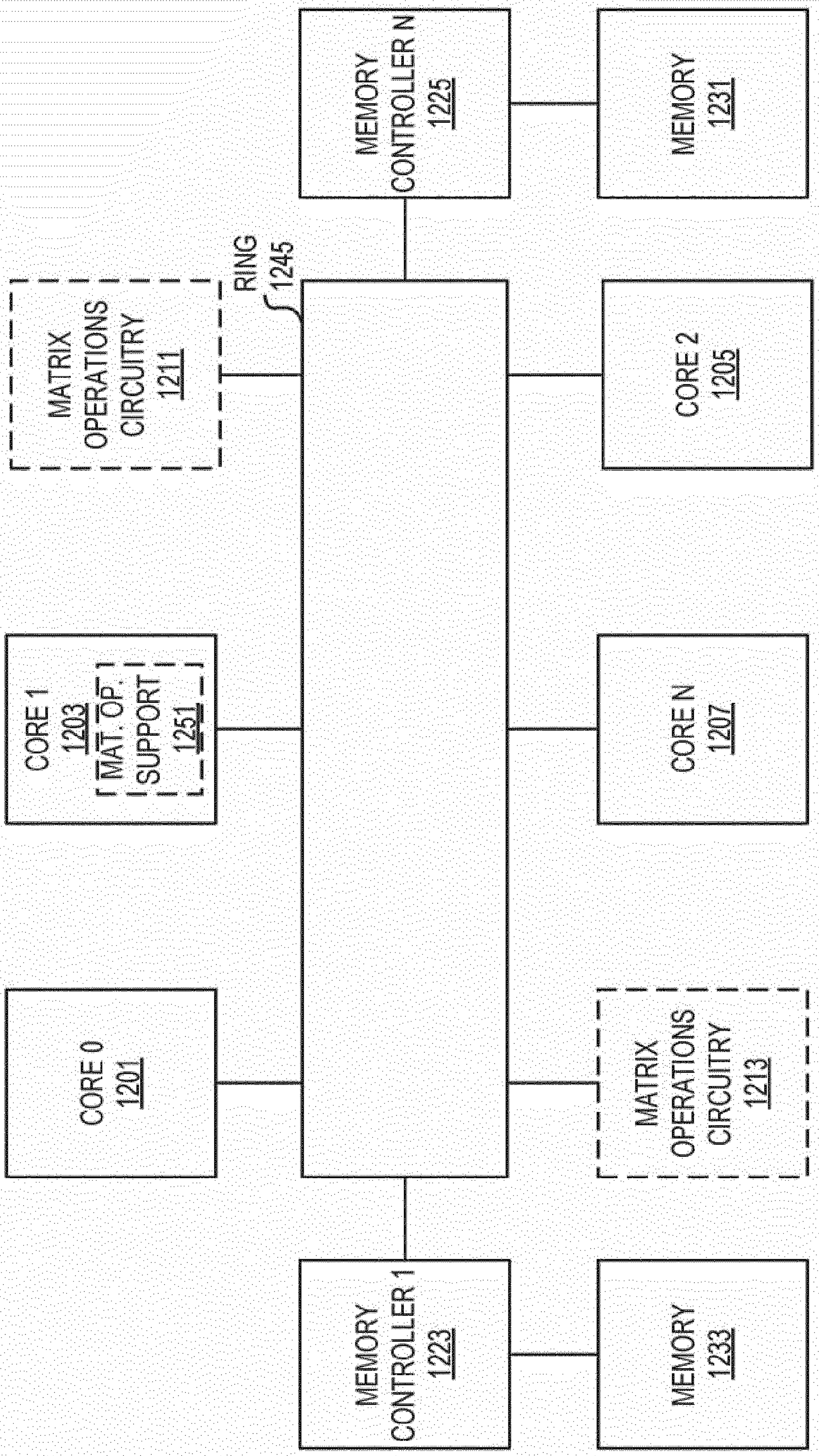

MEMORY CONTROLLER 1 1223
MEMORY 1233
CORE 0 1201
MATRIX OPERATIONS CIRCUITRY 1213
CORE 1 1203
MAT. OP. SUPPORT 1251
CORE N 1207
MATRIX OPERATIONS CIRCUITRY 1211
RING 1245
CORE 2 1205
MEMORY CONTROLLER N 1225
MEMORY 1231

FIG. 12

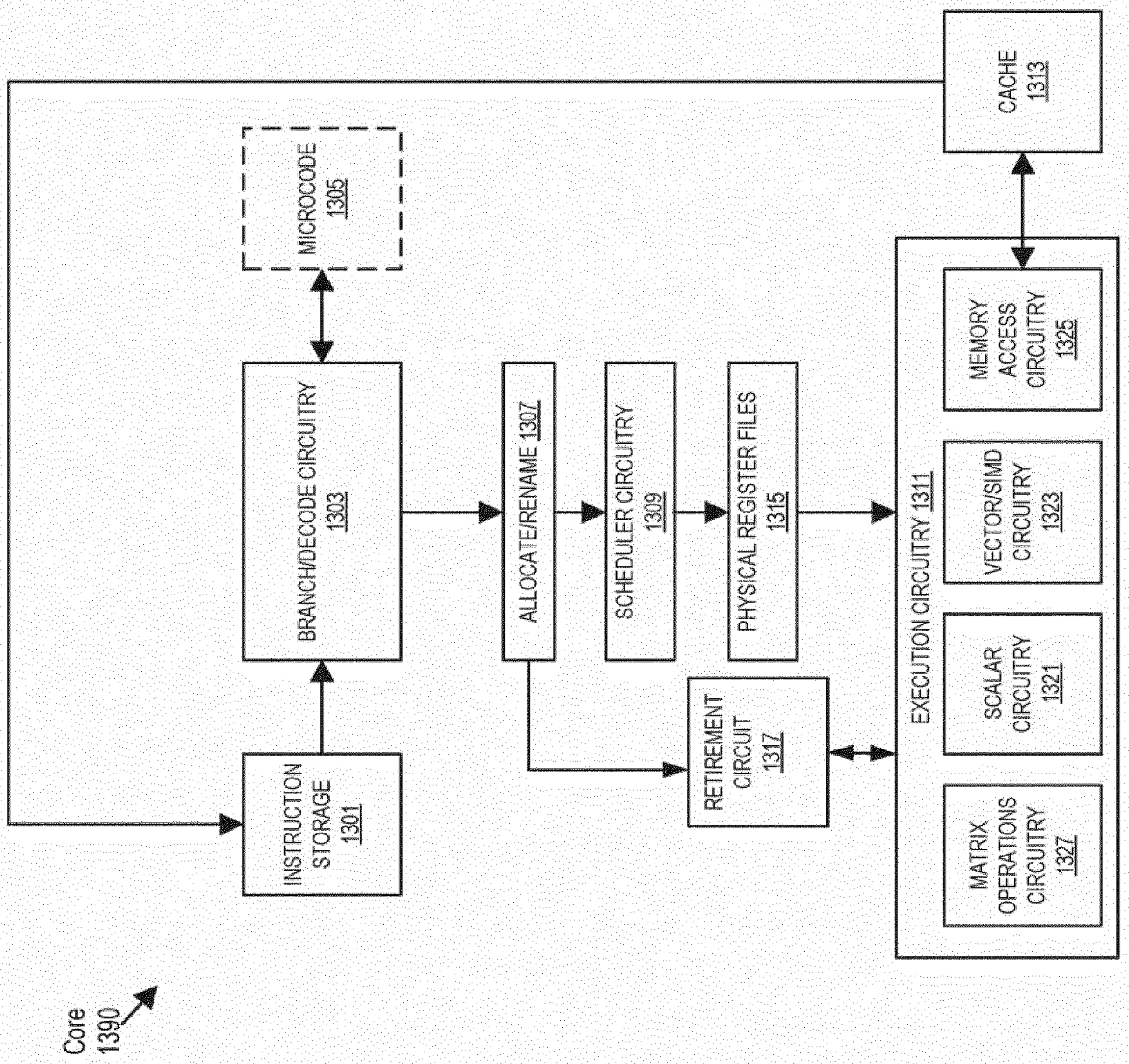

Core 1390
INSTRUCTION STORAGE 1301
BRANCH/DECODE CIRCUITRY 1303
MICROCODE 1305
ALLOCATE/RENAME 1307
SCHEDULER CIRCUITRY 1309
PHYSICAL REGISTER FILES 1315
RETIREMENT CIRCUIT 1317
EXECUTION CIRCUITRY 1311
MATRIX OPERATIONS CIRCUITRY 1327
SCALAR CIRCUITRY 1321
VECTOR/SIMD CIRCUITRY 1323
MEMORY ACCESS CIRCUITRY 1325
CACHE 1313

FIG. 13

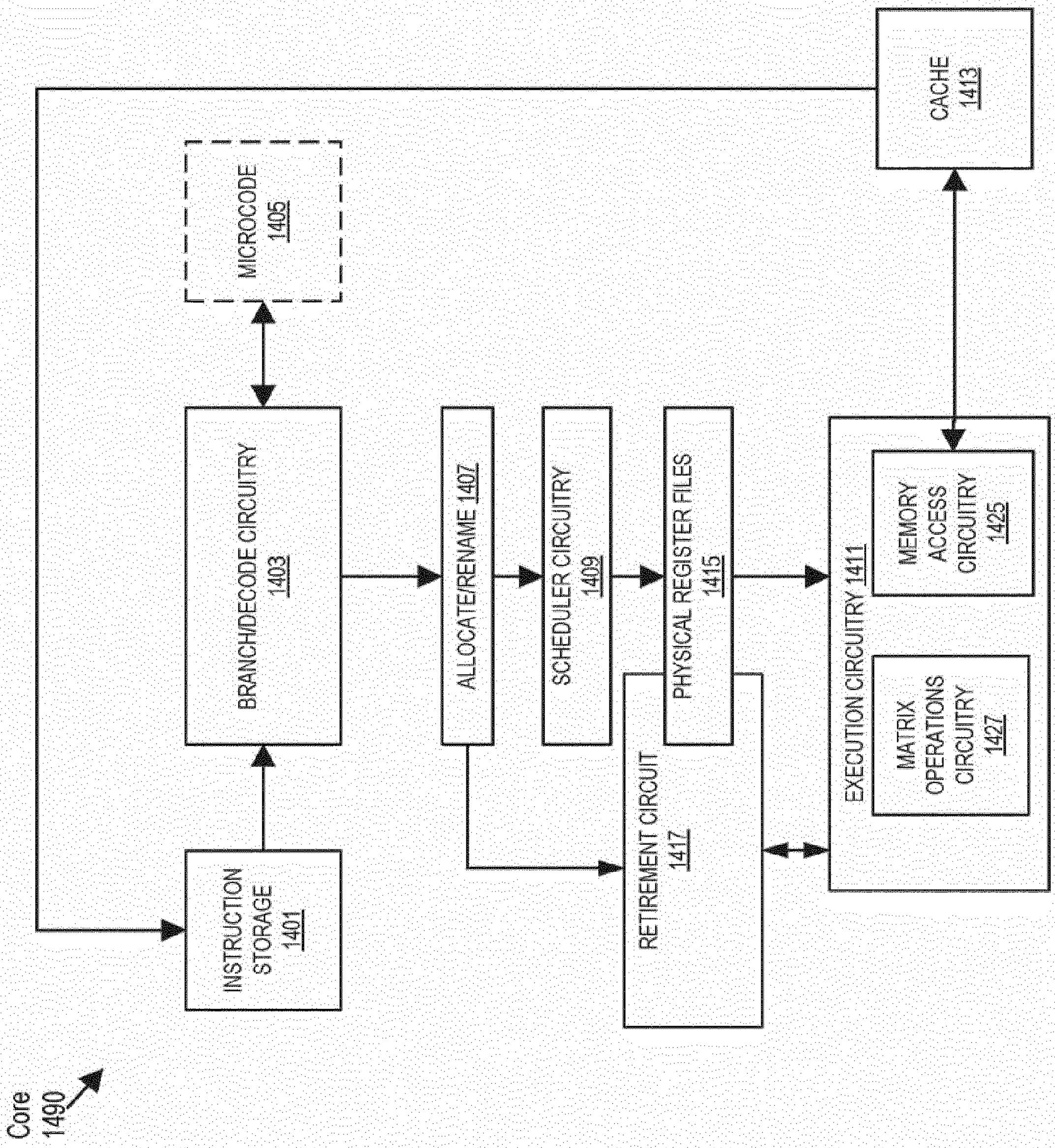

Core 1490
INSTRUCTION STORAGE 1401
BRANCH/DECODE CIRCUITRY 1403
MICROCODE 1405
ALLOCATE/RENAME 1407
SCHEDULER CIRCUITRY 1409
PHYSICAL REGISTER FILES 1415
RETIREMENT CIRCUIT 1417
EXECUTION CIRCUITRY 1411
MATRIX OPERATIONS CIRCUITRY 1427
MEMORY ACCESS CIRCUITRY 1425
CACHE 1413

FIG. 14

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{bmatrix}$$

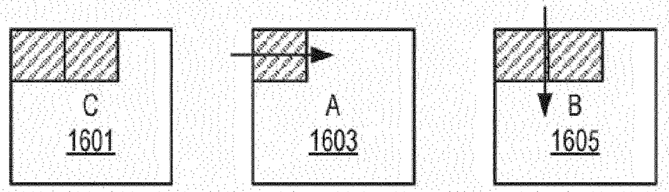
ADDR	VALUE
0	A_{11}
1	A_{12}
2	A_{13}
3	A_{21}
4	A_{22}
5	A_{23}

ROW MAJOR

ADDR	VALUE
0	A_{11}
1	A_{21}
2	A_{12}
3	A_{22}
4	A_{13}
5	A_{23}

COLUMN MAJOR

FIG. 15



```
TILECONFIG [RAX]
// ASSUME SOME OUTER LOOPS DRIVING THE CACHE TILING (NOT SHOWN)
{
TILELOAD TMM0, RSI+RDI // SRCDST, RSI POINTS TO C, RDI HAS
TILELOAD TMM1, RSI+RDI+N // SECOND TILE OF C, UNROLLING IN SIMD DIMENSION N
MOV KK, 0
LOOP:
TILELOAD TMM2, R8+R9 // SRC2 IS STRIDED LOAD OF A, REUSED FOR 2 TMMA INSTR.
TILELOAD TMM3, R10+R11 // SRC1 IS STRIDED LOAD OF B
TMMAPS TMM0, TMM2, TMM3 // UPDATE LEFT TILE OF C
TILELOAD TMM3, R10+R11+N // SRC1 LOADED WITH B FROM NEXT RIGHTMOST TILE
TMMAPS TMM1, TMM2, TMM3 // UPDATE RIGHT TILE OF C
ADD R8, K // UPDATE POINTERS BY CONSTANTS KNOWN OUTSIDE OF LOOP
ADD R10, K*R11
ADD KK, K
CMP KK, LIMIT
JNE LOOP
TILESTORE RSI+RDI, TMM0 // UPDATE THE C MATRIX IN MEMORY
TILESTORE RSI+RDI+M, TMM1
} // END OF OUTER LOOP
TILERELEASE // RETURN TILES TO INIT STATE
```

FIG. 16

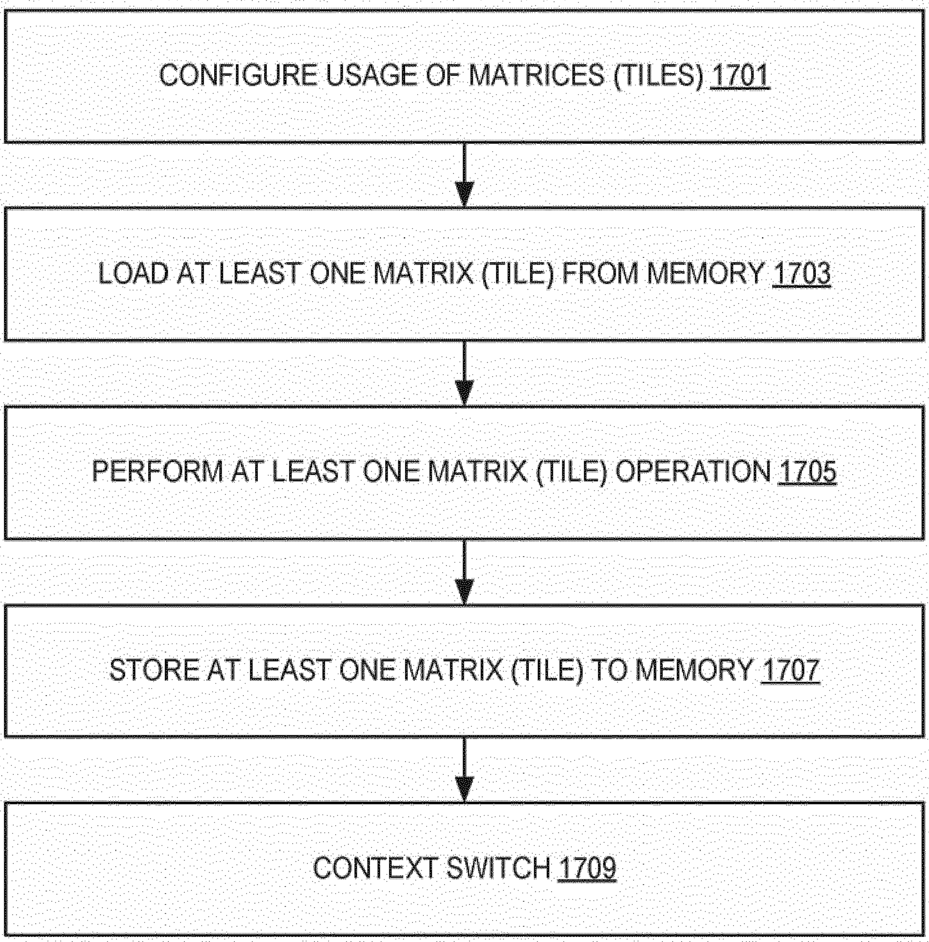

CONFIGURE USAGE OF MATRICES (TILES) 1701
LOAD AT LEAST ONE MATRIX (TILE) FROM MEMORY 1703
PERFORM AT LEAST ONE MATRIX (TILE) OPERATION 1705
STORE AT LEAST ONE MATRIX (TILE) TO MEMORY 1707
CONTEXT SWITCH 1709

FIG. 17

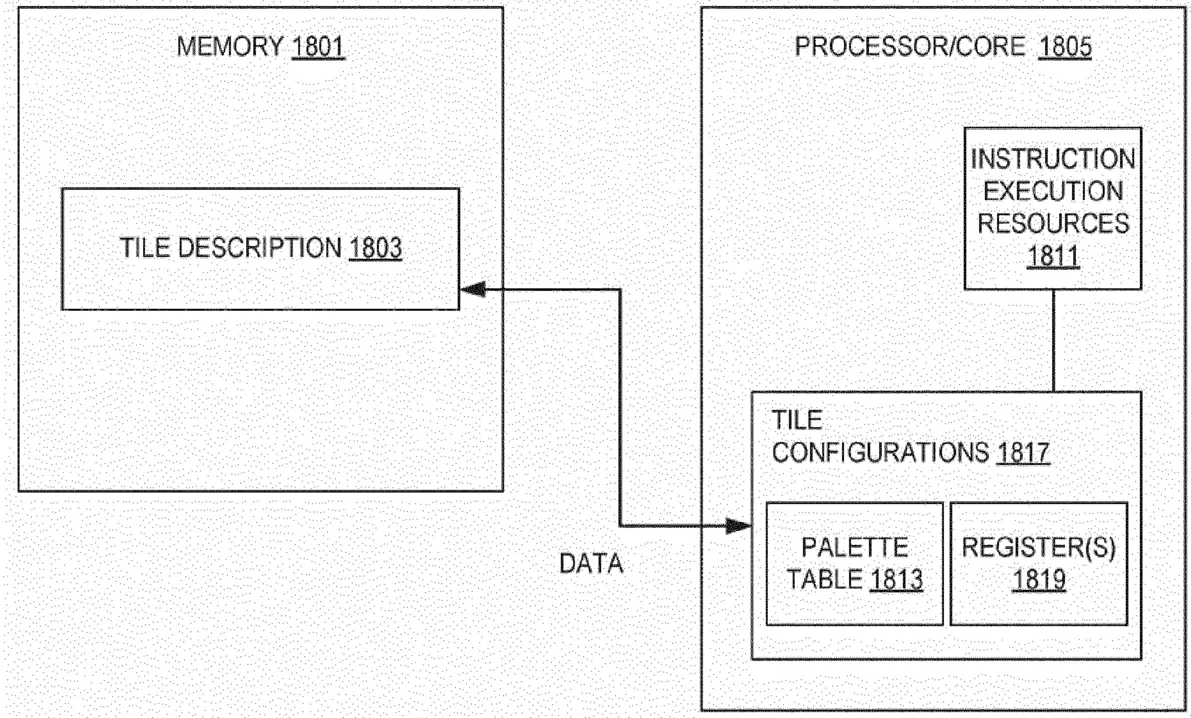

MEMORY 1801
TILE DESCRIPTION 1803
PROCESSOR/CORE 1805
INSTRUCTION EXECUTION RESOURCES 1811
TILE CONFIGURATIONS 1817
PALETTE TABLE 1813
REGISTER(S) 1819
DATA

FIG. 18

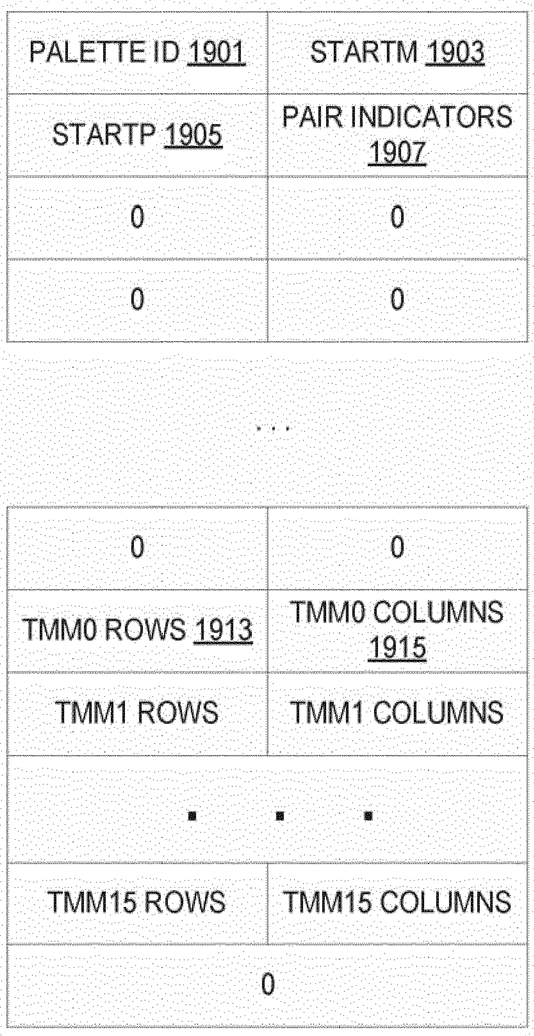

PALETTE ID 1901
STARTM 1903
STARTP 1905
PAIR INDICATORS 1907
0
0
0
0
0
0
TMM0 ROWS 1913
TMM0 COLUMNS 1915
TMM1 ROWS
TMM1 COLUMNS
TMM15 ROWS
TMM15 COLUMNS
0

FIG. 19

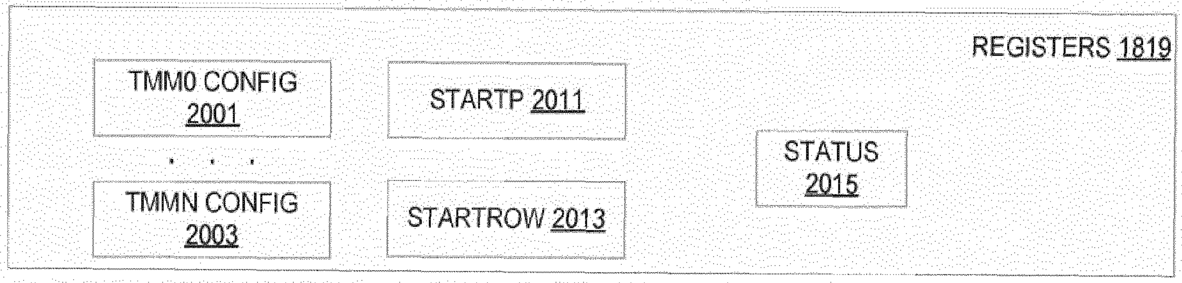


FIG. 20(A)

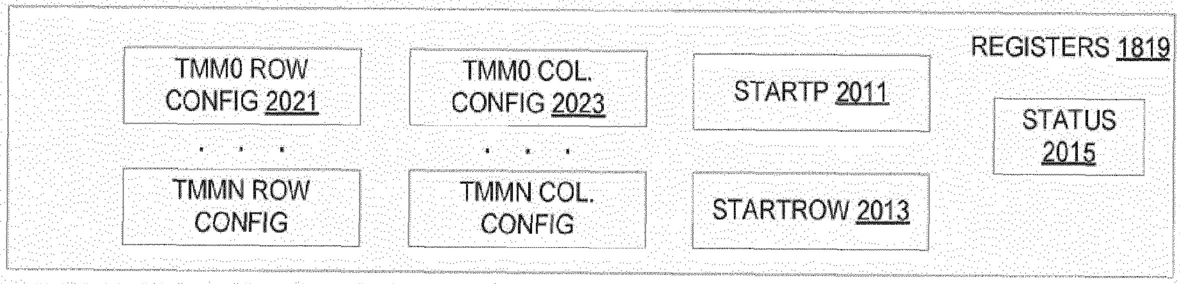


FIG. 20(B)

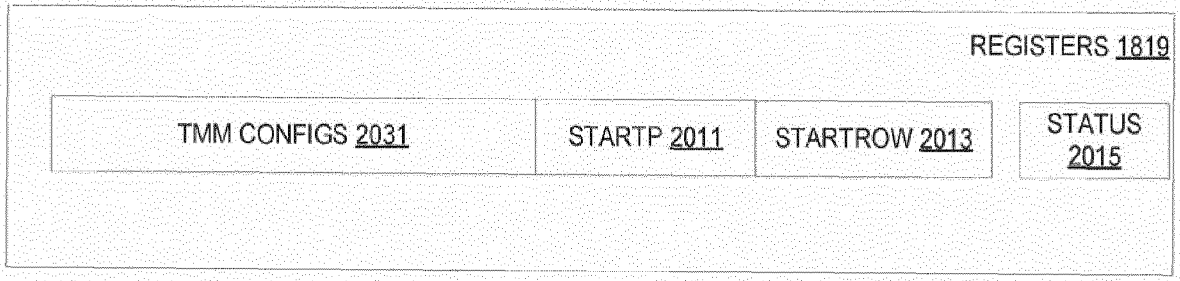


FIG. 20(C)

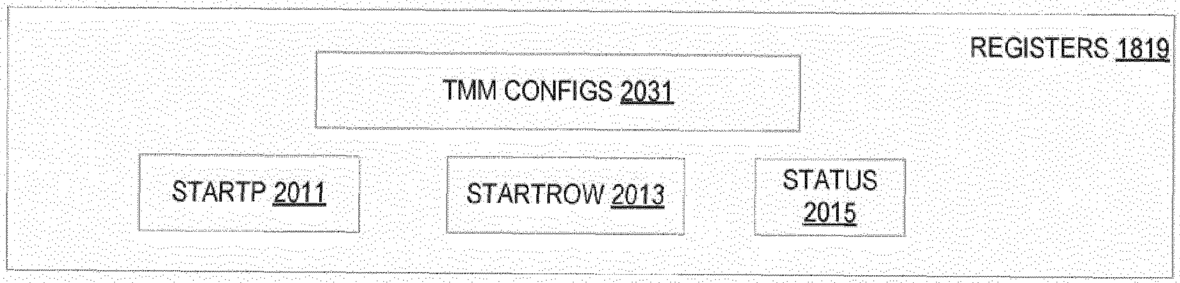


FIG. 20(D)

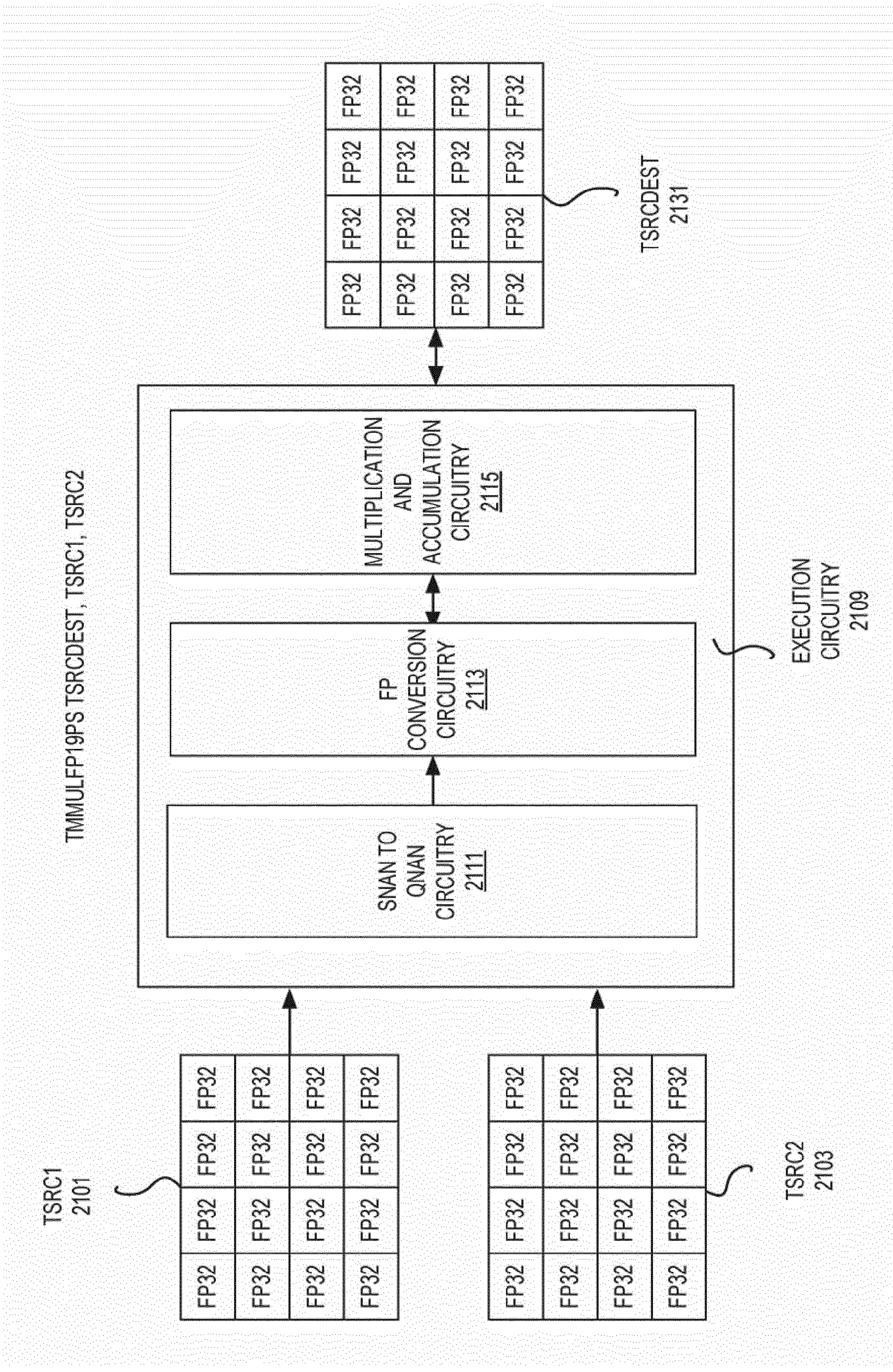

TMMULFP19PS TSRCDEST, TSRC1, TSRC2
TSRC1
2101
FP32
TSRC2
2103
SNAN TO QNAN CIRCUITRY 2111
FP CONVERSION CIRCUITRY 2113
MULTIPLICATION AND ACCUMULATION CIRCUITRY 2115
EXECUTION CIRCUITRY 2109
TSRCDEST
2131

FIG. 21

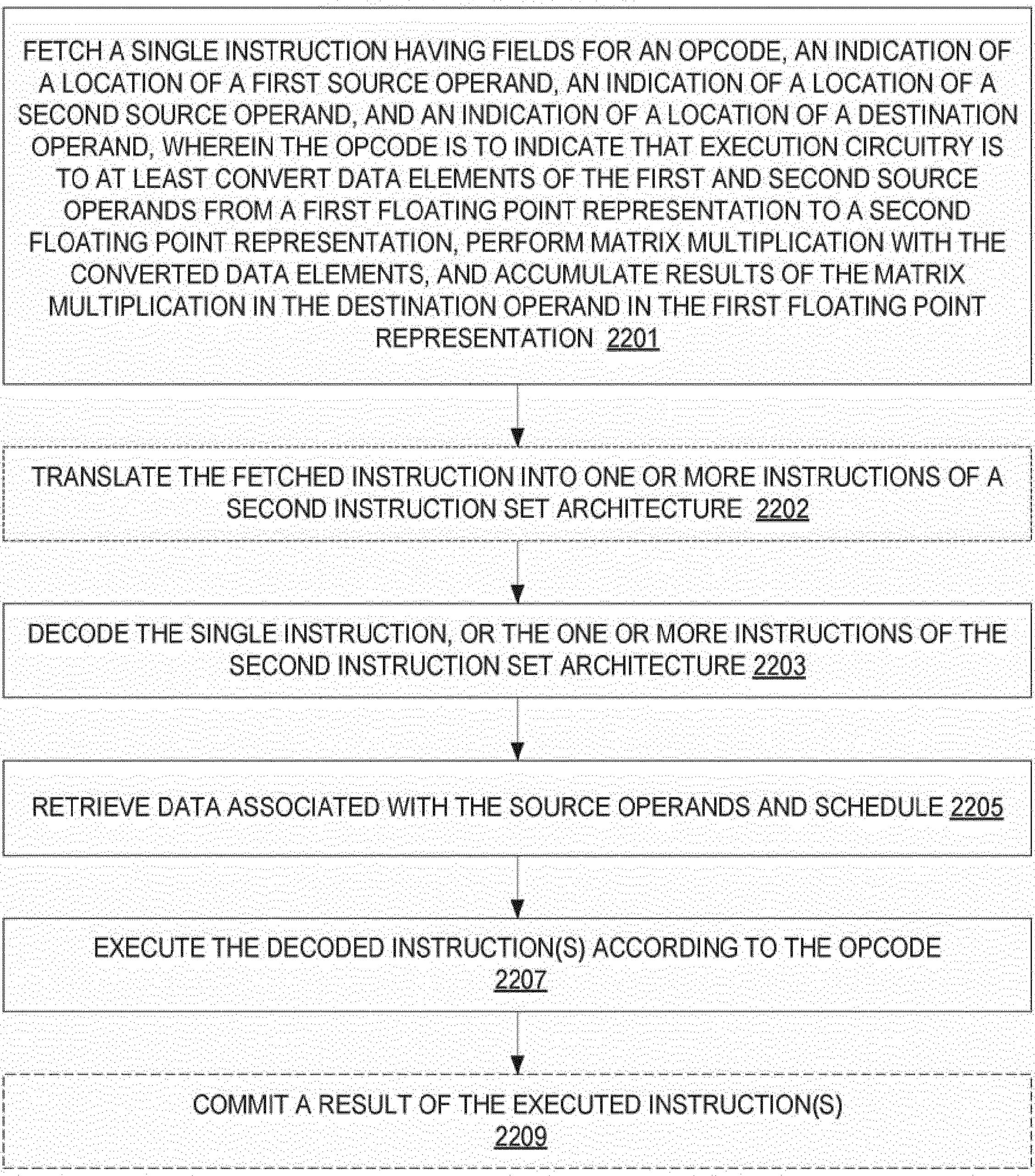

FETCH A SINGLE INSTRUCTION HAVING FIELDS FOR AN OPCODE, AN INDICATION OF A LOCATION OF A FIRST SOURCE OPERAND, AN INDICATION OF A LOCATION OF A SECOND SOURCE OPERAND, AND AN INDICATION OF A LOCATION OF A DESTINATION OPERAND, WHEREIN THE OPCODE IS TO INDICATE THAT EXECUTION CIRCUITRY IS TO AT LEAST CONVERT DATA ELEMENTS OF THE FIRST AND SECOND SOURCE OPERANDS FROM A FIRST FLOATING POINT REPRESENTATION TO A SECOND FLOATING POINT REPRESENTATION, PERFORM MATRIX MULTIPLICATION WITH THE CONVERTED DATA ELEMENTS, AND ACCUMULATE RESULTS OF THE MATRIX MULTIPLICATION IN THE DESTINATION OPERAND IN THE FIRST FLOATING POINT REPRESENTATION 2201
TRANSLATE THE FETCHED INSTRUCTION INTO ONE OR MORE INSTRUCTIONS OF A SECOND INSTRUCTION SET ARCHITECTURE 2202
DECODE THE SINGLE INSTRUCTION, OR THE ONE OR MORE INSTRUCTIONS OF THE SECOND INSTRUCTION SET ARCHITECTURE 2203
RETRIEVE DATA ASSOCIATED WITH THE SOURCE OPERANDS AND SCHEDULE 2205
EXECUTE THE DECODED INSTRUCTION(S) ACCORDING TO THE OPCODE 2207
COMMIT A RESULT OF THE EXECUTED INSTRUCTION(S) 2209

FIG. 22

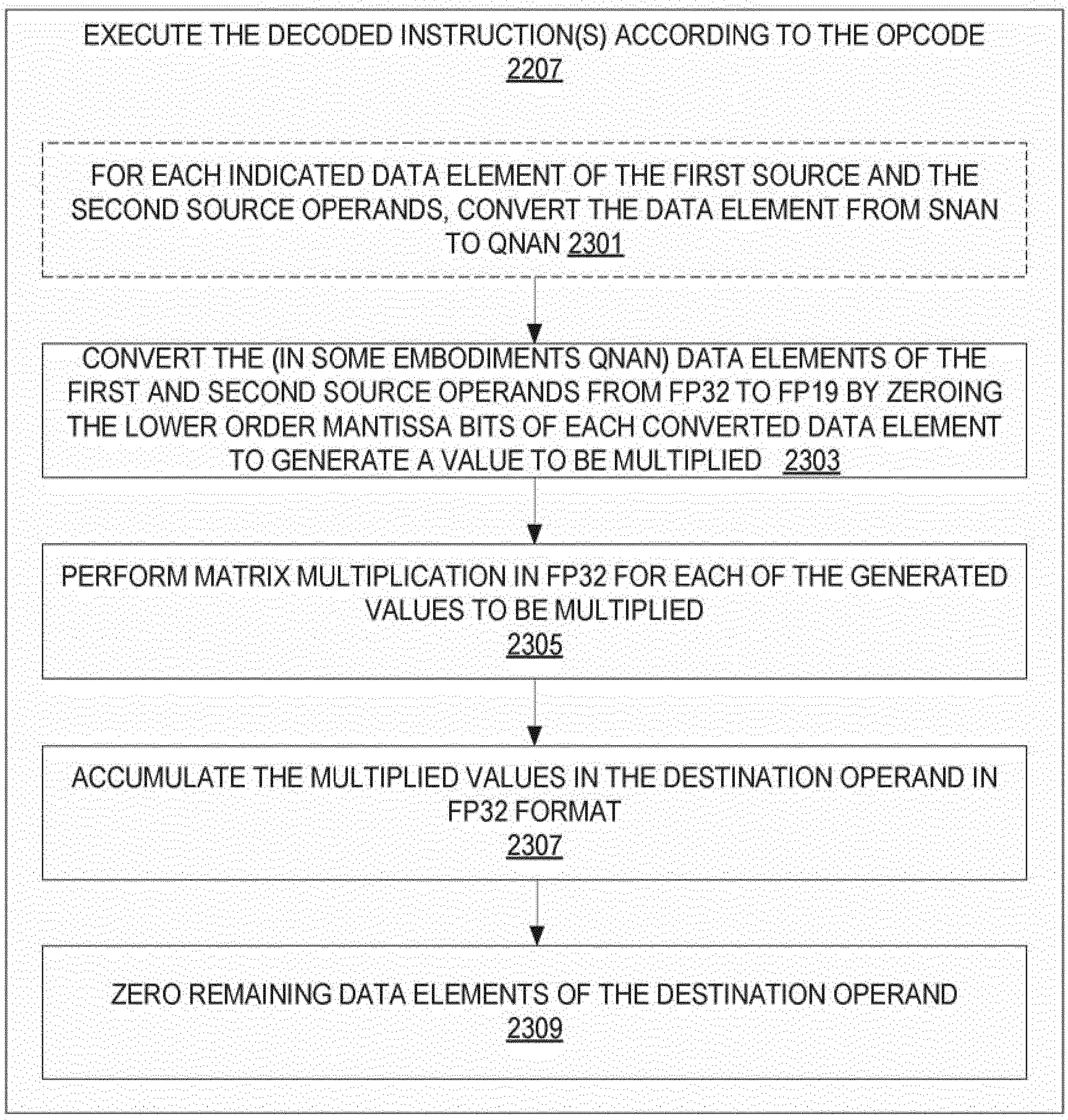

EXECUTE THE DECODED INSTRUCTION(S) ACCORDING TO THE OPCODE 2207
FOR EACH INDICATED DATA ELEMENT OF THE FIRST SOURCE AND THE SECOND SOURCE OPERANDS, CONVERT THE DATA ELEMENT FROM SNAN TO QNAN 2301
CONVERT THE (IN SOME EMBODIMENTS QNAN) DATA ELEMENTS OF THE FIRST AND SECOND SOURCE OPERANDS FROM FP32 TO FP19 BY ZEROING THE LOWER ORDER MANTISSA BITS OF EACH CONVERTED DATA ELEMENT TO GENERATE A VALUE TO BE MULTIPLIED 2303
PERFORM MATRIX MULTIPLICATION IN FP32 FOR EACH OF THE GENERATED VALUES TO BE MULTIPLIED 2305
ACCUMULATE THE MULTIPLIED VALUES IN THE DESTINATION OPERAND IN FP32 FORMAT 2307
ZERO REMAINING DATA ELEMENTS OF THE DESTINATION OPERAND 2309

FIG. 23

```
Define Zero_lower_mantissa_bits_fp32(x):
// The Input Is A FP32 Number, The Output Is A FP32 Number With The Low 13 Bits Set To Zero
// This Masked FP32 Is Referred To As FP19
        Dword := 0   //dword Is 32-Bit
        Dword[31:13] := X[31:13]
        Return Dword

Define Silence_snan_fp32(x):
// SNAN Has Exponent 255 And High Fraction Bit Set To 0 And Non-Zero Other Fraction Bits
        If X.exponent == 255 And X.fraction != 0 And X.fraction[22] == 0:
                X.fraction[22] := 1
        Return X



TMMULFP19PS Tsrcdest, Tsrc1, Tsrc2
// C = M X N (tsrcdest), A – M X K (tsrc1), B = K X N (tsrc2), Tsrc1 And Tsrc2 Elements Are FP32
Elements_src1 := Tsrc1.colsb / 4
Elements_src2 := Tsrc2.colsb / 4
Elements_dest := Tsrcdest.colsb / 4 //colsb = Provided Bytes Per Row
Elements_temp := Tsrcdest.colsb / 4

For M In 0 ... Tsrcdest.rows-1:
        // SET TEMP1 HAVING A NUMBER OF ELEMENTS =ELEMENTS_TEMP SIZE
        Temp1[ 0 ... Elements_temp-1 ] := 0
        // FOR K FROM 0 TO NUMBER OF ELEMENTS IN SRC1-1
        For K In 0 ... Elements_src1-1:
                // FOR N FROM 0 TO NUMBER OF ELEMENTS IN SRCDEST-1
                For N In 0 ... Elements_dest-1:
                        //FP32 FMA With DAZ=FTZ=1, RNE Rounding
                        // MXCSR Is Neither Consulted Nor Updated, No Exceptions Raised Or
Denoted
                        A := Silence_snan_fp32(tsrc1.row[m].fp32[k])
                        B := Silence_snan_fp32(tsrc2.row[k].fp32[n])
                        Temp1.fp32[n] += Zero_lower_mantissa_bits_fp32(a) *
                                         Zero_lower_mantissa_bits_fp32(b)
        For N In 0 ... Elements_dest-1:
                //FP32 FMA With DAZ=FTZ=1, RNE Rounding, MXCSR Is Neither Consulted Nor
Updated
                // No Exceptions Raised Or Denoted
                Tsrcdest.row[m].fp32[n] := Tsrcdest.row[m].fp32[n] + Temp1.fp32[n]
        Write_row_and_zero(tsrcdest, M, Tmp, Tsrcdest.colsb)
Zero_upper_rows(tsrcdest, Tsrcdest.rows)
Zero_tileconfig_start()
```

FIG. 24

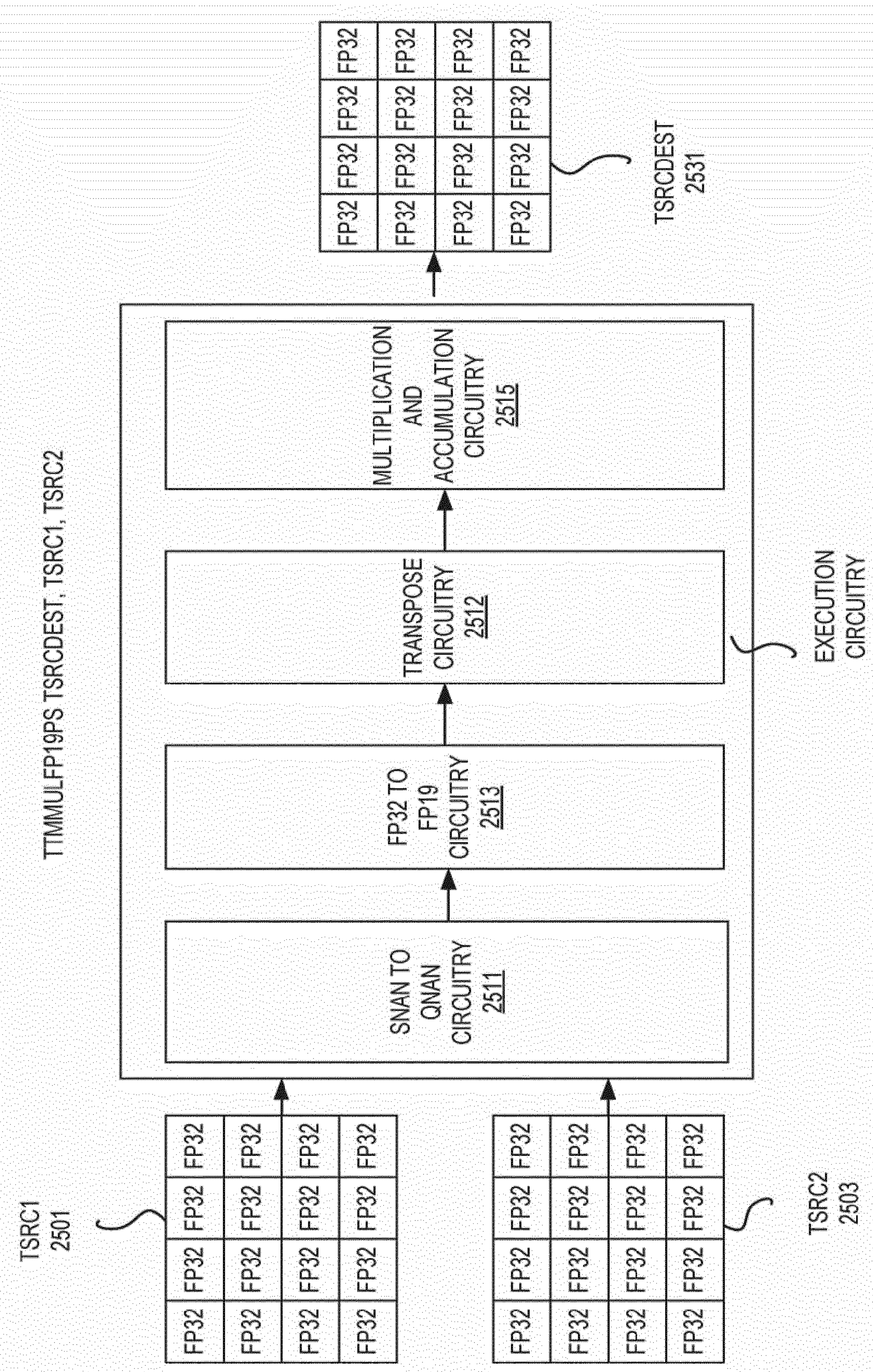

TTMMULFP19PS TSRCDEST, TSRC1, TSRC2
TSRC1 2501
TSRC2 2503
FP32
SNAN TO QNAN CIRCUITRY 2511
FP32 TO FP19 CIRCUITRY 2513
TRANSPOSE CIRCUITRY 2512
MULTIPLICATION AND ACCUMULATION CIRCUITRY 2515
EXECUTION CIRCUITRY 2509
TSRCDEST 2531

FIG. 25

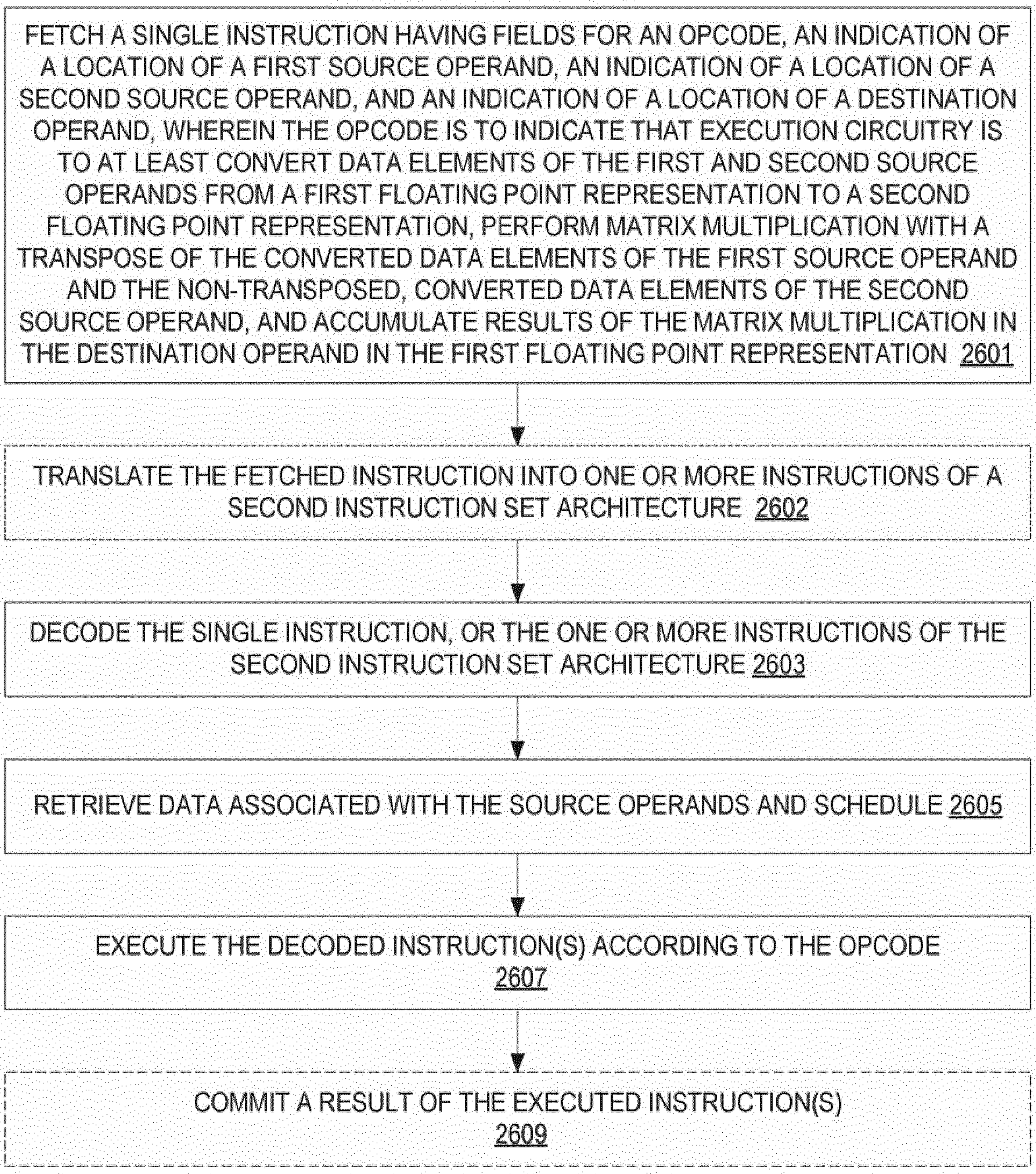

FETCH A SINGLE INSTRUCTION HAVING FIELDS FOR AN OPCODE, AN INDICATION OF A LOCATION OF A FIRST SOURCE OPERAND, AN INDICATION OF A LOCATION OF A SECOND SOURCE OPERAND, AND AN INDICATION OF A LOCATION OF A DESTINATION OPERAND, WHEREIN THE OPCODE IS TO INDICATE THAT EXECUTION CIRCUITRY IS TO AT LEAST CONVERT DATA ELEMENTS OF THE FIRST AND SECOND SOURCE OPERANDS FROM A FIRST FLOATING POINT REPRESENTATION TO A SECOND FLOATING POINT REPRESENTATION, PERFORM MATRIX MULTIPLICATION WITH A TRANSPOSE OF THE CONVERTED DATA ELEMENTS OF THE FIRST SOURCE OPERAND AND THE NON-TRANSPOSED, CONVERTED DATA ELEMENTS OF THE SECOND SOURCE OPERAND, AND ACCUMULATE RESULTS OF THE MATRIX MULTIPLICATION IN THE DESTINATION OPERAND IN THE FIRST FLOATING POINT REPRESENTATION 2601
TRANSLATE THE FETCHED INSTRUCTION INTO ONE OR MORE INSTRUCTIONS OF A SECOND INSTRUCTION SET ARCHITECTURE 2602
DECODE THE SINGLE INSTRUCTION, OR THE ONE OR MORE INSTRUCTIONS OF THE SECOND INSTRUCTION SET ARCHITECTURE 2603
RETRIEVE DATA ASSOCIATED WITH THE SOURCE OPERANDS AND SCHEDULE 2605
EXECUTE THE DECODED INSTRUCTION(S) ACCORDING TO THE OPCODE 2607
COMMIT A RESULT OF THE EXECUTED INSTRUCTION(S) 2609

FIG. 26

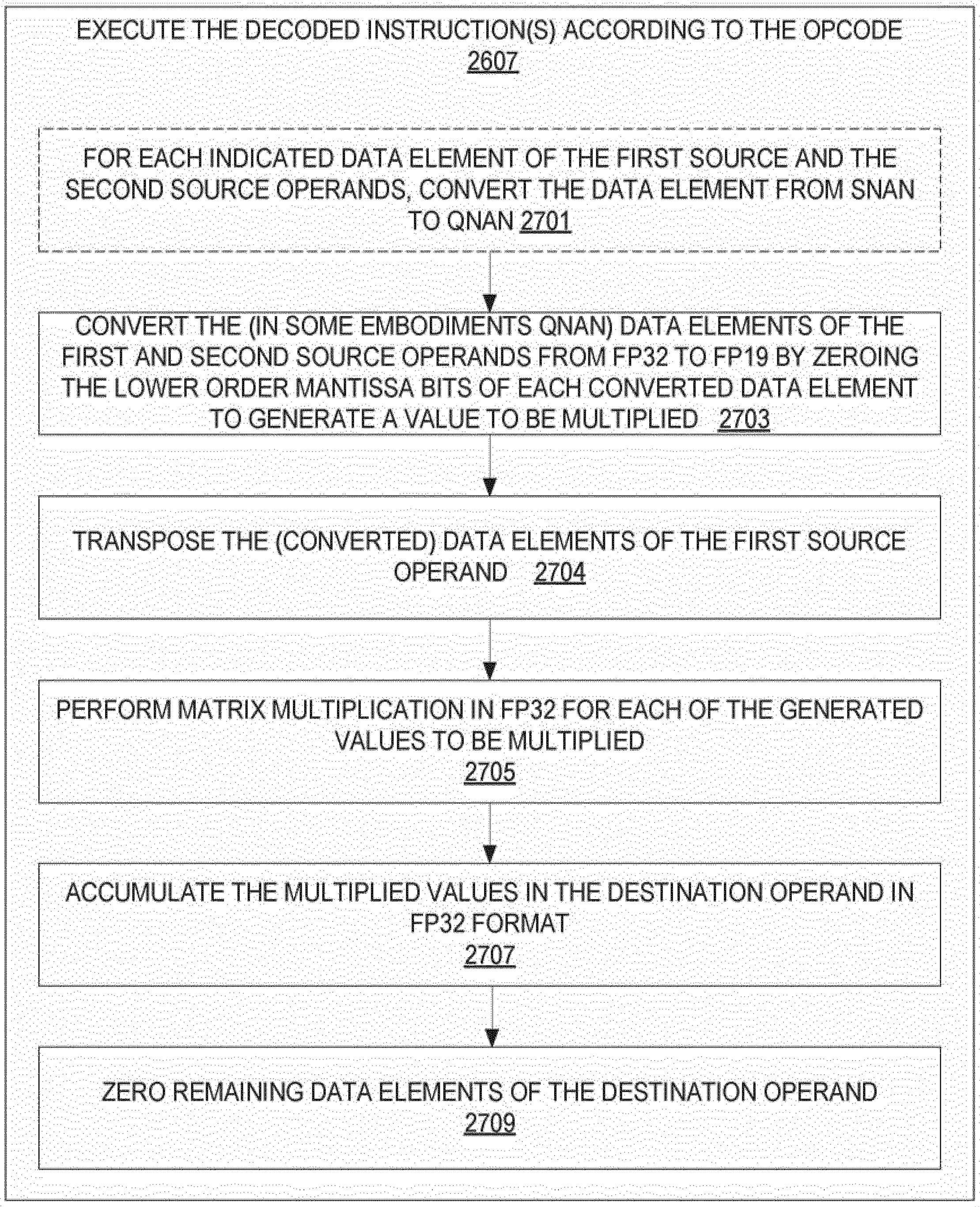

EXECUTE THE DECODED INSTRUCTION(S) ACCORDING TO THE OPCODE 2607
FOR EACH INDICATED DATA ELEMENT OF THE FIRST SOURCE AND THE SECOND SOURCE OPERANDS, CONVERT THE DATA ELEMENT FROM SNAN TO QNAN 2701
CONVERT THE (IN SOME EMBODIMENTS QNAN) DATA ELEMENTS OF THE FIRST AND SECOND SOURCE OPERANDS FROM FP32 TO FP19 BY ZEROING THE LOWER ORDER MANTISSA BITS OF EACH CONVERTED DATA ELEMENT TO GENERATE A VALUE TO BE MULTIPLIED 2703
TRANSPOSE THE (CONVERTED) DATA ELEMENTS OF THE FIRST SOURCE OPERAND 2704
PERFORM MATRIX MULTIPLICATION IN FP32 FOR EACH OF THE GENERATED VALUES TO BE MULTIPLIED 2705
ACCUMULATE THE MULTIPLIED VALUES IN THE DESTINATION OPERAND IN FP32 FORMAT 2707
ZERO REMAINING DATA ELEMENTS OF THE DESTINATION OPERAND 2709

FIG. 27

```
Define Zero_lower_mantissa_bits_fp32(x):
// The Input Is A FP32 Number, The Output Is A FP32 Number With The Low 13 Bits Set To Zero
// This Masked FP32 Is Referred To As FP19
        Dword := 0   //dword Is 32-Bit
        Dword[31:13] := X[31:13]
        Return Dword

Define Silence_sna_fp32(x):
// SNAN Has Exponent 255 And High Fraction Bit Set To 0 And Non-Zero Other Fraction Bits
        If X.exponent == 255 And X.fraction != 0 And X.fraction[22] == 0:
                X.fraction[22] := 1
        Return X


TTMMULFP19PS Tsrcdest, Tsrc1, Tsrc2
// C = M X N (tsrcdest), A – K X M (tsrc1), B = K X N (tsrc2)
// Src1 And Src2 Elements Are FP32
Elements_dest := Tsrcdest.colsb / 4
Elements_temp := Tsrcdest.colsb / 4

For M In 0 ... Tsrcdest.rows-1:
        Temp1[ 0 ... Elements_temp-1 ] := 0
        For K In 0 ... Tsrc1.rows-1:
                For N In 0 ... Elements_dest-1:
                        A1e := Silence_snan_fp32(tsrc1.row[k].fp32[m])
                        A2e := Silence_snan_fp32(tsrc2.row[k].fp32[n])
                        S1e :=  Zero_lower_mantissa_bits_fp32(a1e)
                        S2e := Zero_lower_mantissa_bits_fp32(a2e)
                //FP32 FMA With DAZ=FTZ=1, RNE Rounding
                // MXCSR Is Neither Consulted Nor Updated
                // No Exceptions Raised Or Denoted
                        Temp1.fp32[n] += Fma32(temp1.fp32[n], S1e, S2e, Daz=1, Ftz=1, Sae=1,
Rc=RNE)
        For N In 0 ... Elements_dest-1:
                //FP32 FMA With DAZ=FTZ=1, RNE Rounding
                // MXCSR Is Neither Consulted Nor Updated
                // No Exceptions Raised Or Denoted
                Tsrcdest.row[m].fp32[n] := Srcdest.row[m].fp32[n] + Temp1.fp32[n]
        Write_row_and_zero(tsrcdest, M, Tmp, Tsrcdest.colsb)

Zero_upper_rows(tsrcdest, Tsrcdest.rows)
Zero_tileconfig_start()
```

FIG. 28

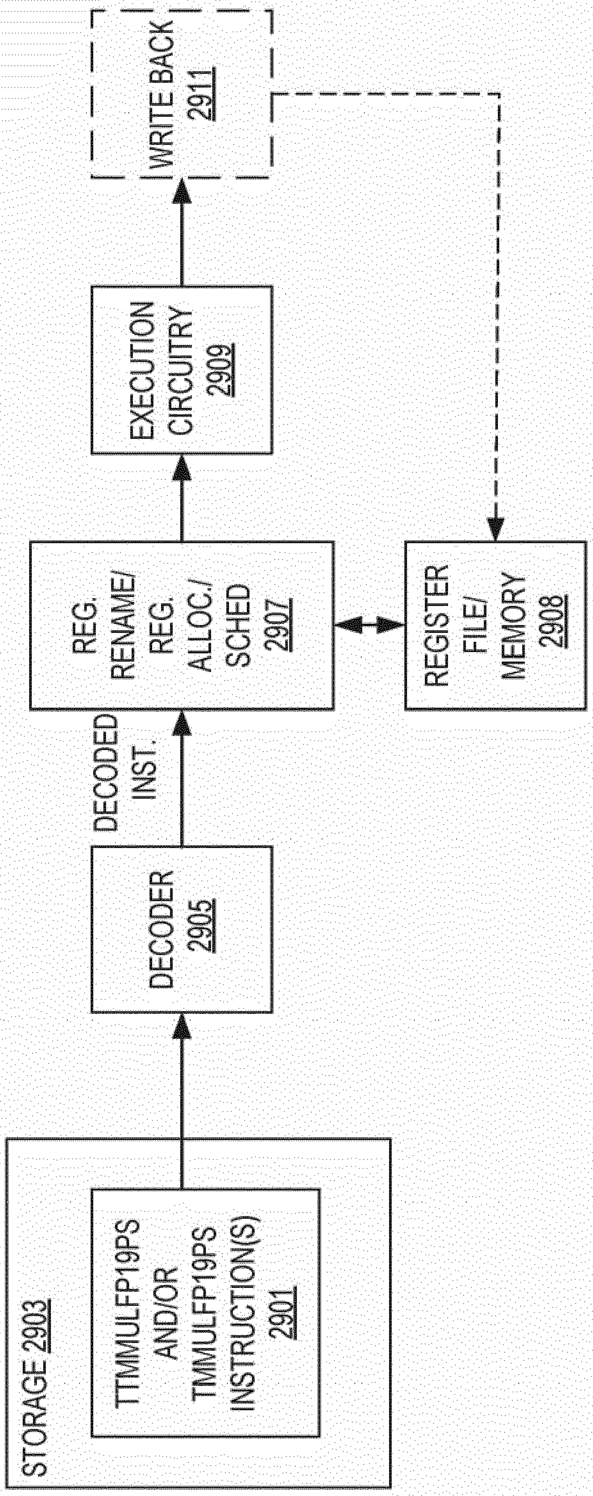

STORAGE 2903
TTMMULFP19PS AND/OR TMMULFP19PS INSTRUCTION(S) 2901
DECODER 2905
DECODED INST.
REG. RENAME/ REG. ALLOC./ SCHED 2907
REGISTER FILE/ MEMORY 2908
EXECUTION CIRCUITRY 2909
WRITE BACK 2911

FIG. 29

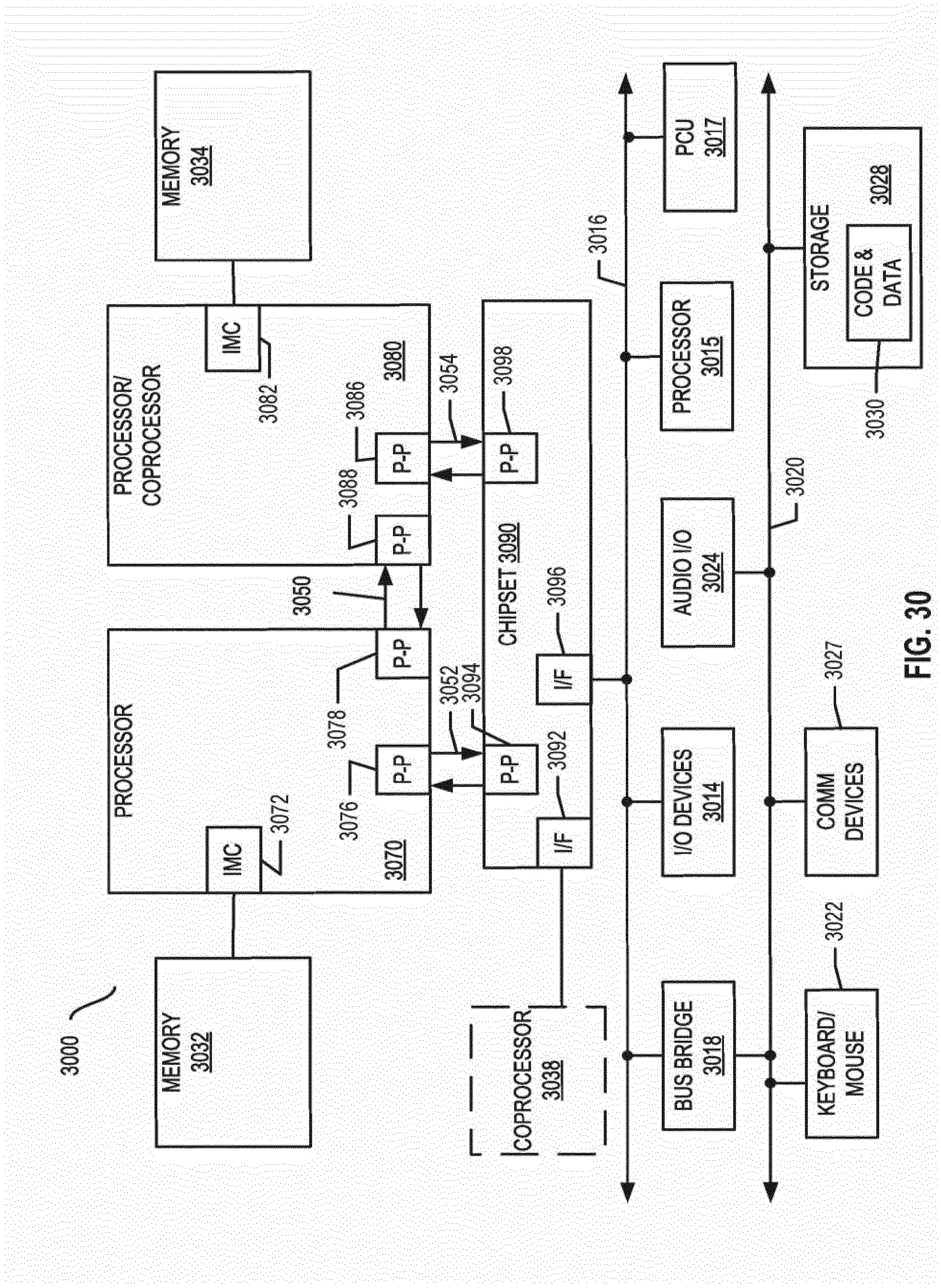

3000
MEMORY 3032
MEMORY 3034
PROCESSOR
IMC
3072
3076
P-P
3078
P-P
3070
PROCESSOR/ COPROCESSOR
IMC
3082
3086
P-P
3088
P-P
3080
3050
3052
3094
3054
3098
P-P
P-P
CHIPSET 3090
3092
I/F
3096
I/F
COPROCESSOR 3038
3016
BUS BRIDGE 3018
I/O DEVICES 3014
AUDIO I/O 3024
PROCESSOR 3015
PCU 3017
3020
KEYBOARD/ MOUSE
3022
COMM DEVICES
3027
STORAGE 3028
CODE & DATA
3030

FIG. 30

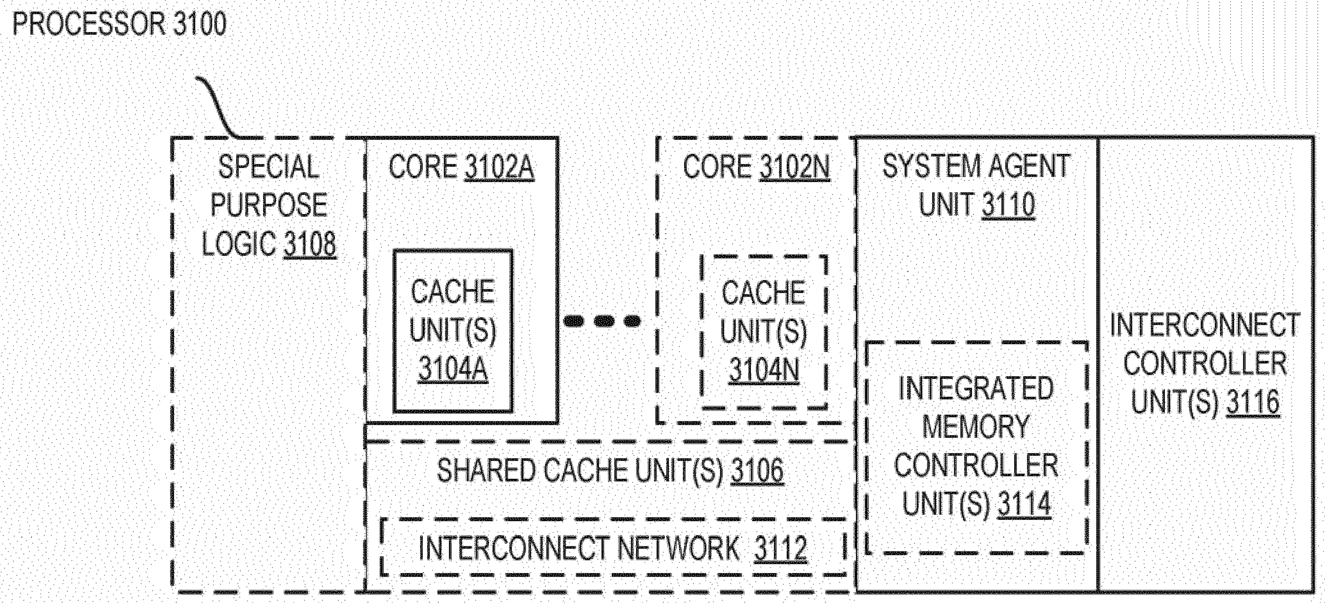

PROCESSOR 3100
SPECIAL PURPOSE LOGIC 3108
CORE 3102A
CACHE UNIT(S) 3104A
CORE 3102N
CACHE UNIT(S) 3104N
SYSTEM AGENT UNIT 3110
INTEGRATED MEMORY CONTROLLER UNIT(S) 3114
INTERCONNECT CONTROLLER UNIT(S) 3116
SHARED CACHE UNIT(S) 3106
INTERCONNECT NETWORK 3112

FIG. 31

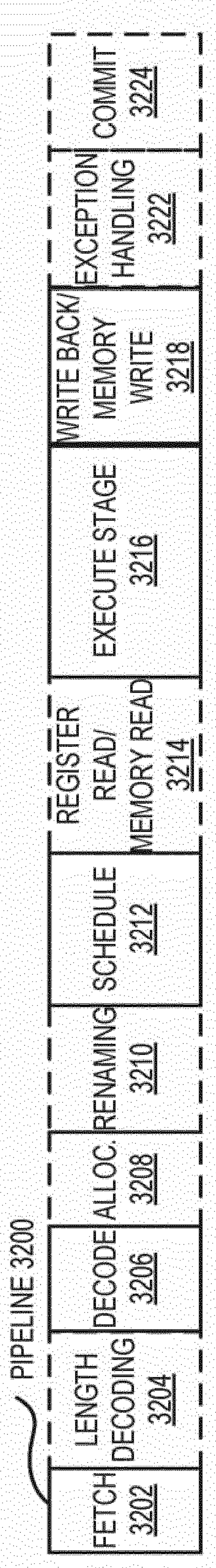

PIPELINE 3200
FETCH 3202
LENGTH DECODING 3204
DECODE 3206
ALLOC. 3208
RENAMING 3210
SCHEDULE 3212
REGISTER READ/ MEMORY READ 3214
EXECUTE STAGE 3216
WRITE BACK/ MEMORY WRITE 3218
EXCEPTION HANDLING 3222
COMMIT 3224

FIG. 32(A)

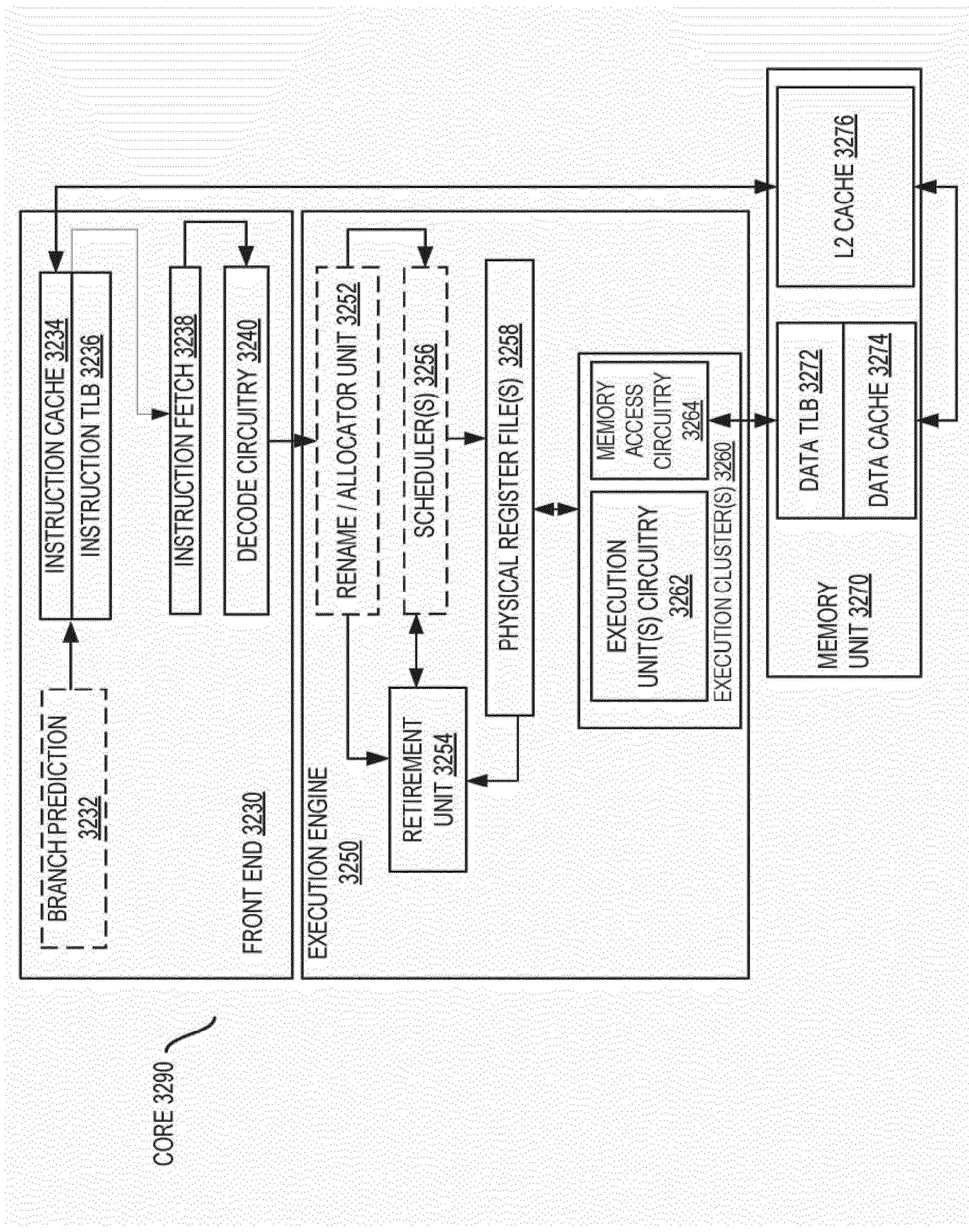

CORE 3290
FRONT END 3230
BRANCH PREDICTION 3232
INSTRUCTION CACHE 3234
INSTRUCTION TLB 3236
INSTRUCTION FETCH 3238
DECODE CIRCUITRY 3240
EXECUTION ENGINE 3250
RENAME / ALLOCATOR UNIT 3252
SCHEDULER(S) 3256
RETIREMENT UNIT 3254
PHYSICAL REGISTER FILE(S) 3258
EXECUTION UNIT(S) CIRCUITRY 3262
MEMORY ACCESS CIRCUITRY 3264
EXECUTION CLUSTER(S) 3260
MEMORY UNIT 3270
DATA TLB 3272
DATA CACHE 3274
L2 CACHE 3276

FIG. 32(B)

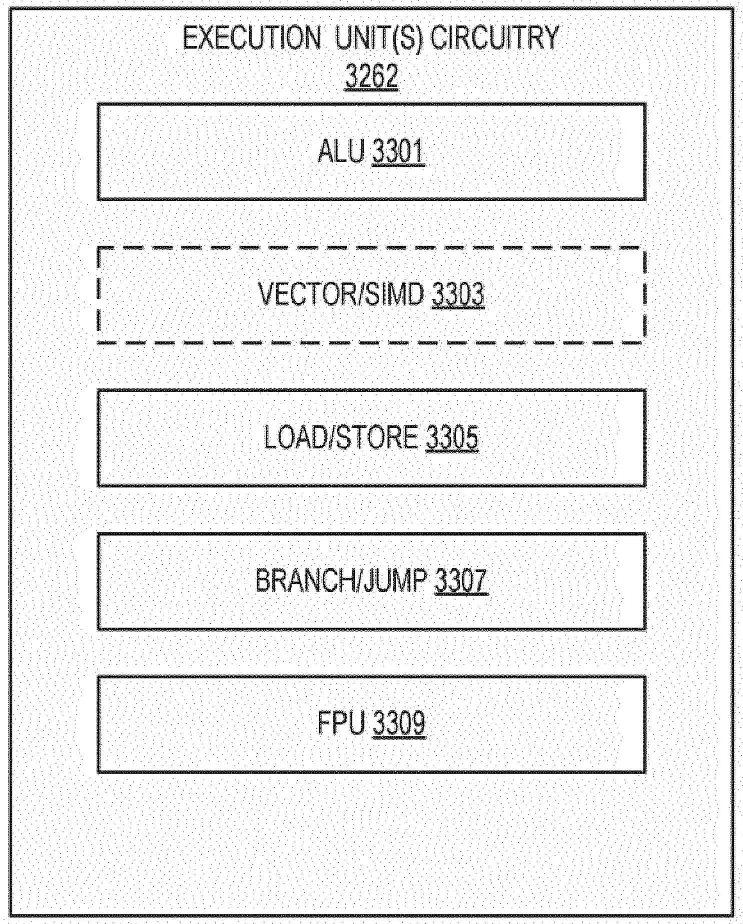

EXECUTION UNIT(S) CIRCUITRY
3262
ALU 3301
VECTOR/SIMD 3303
LOAD/STORE 3305
BRANCH/JUMP 3307
FPU 3309

FIG. 33

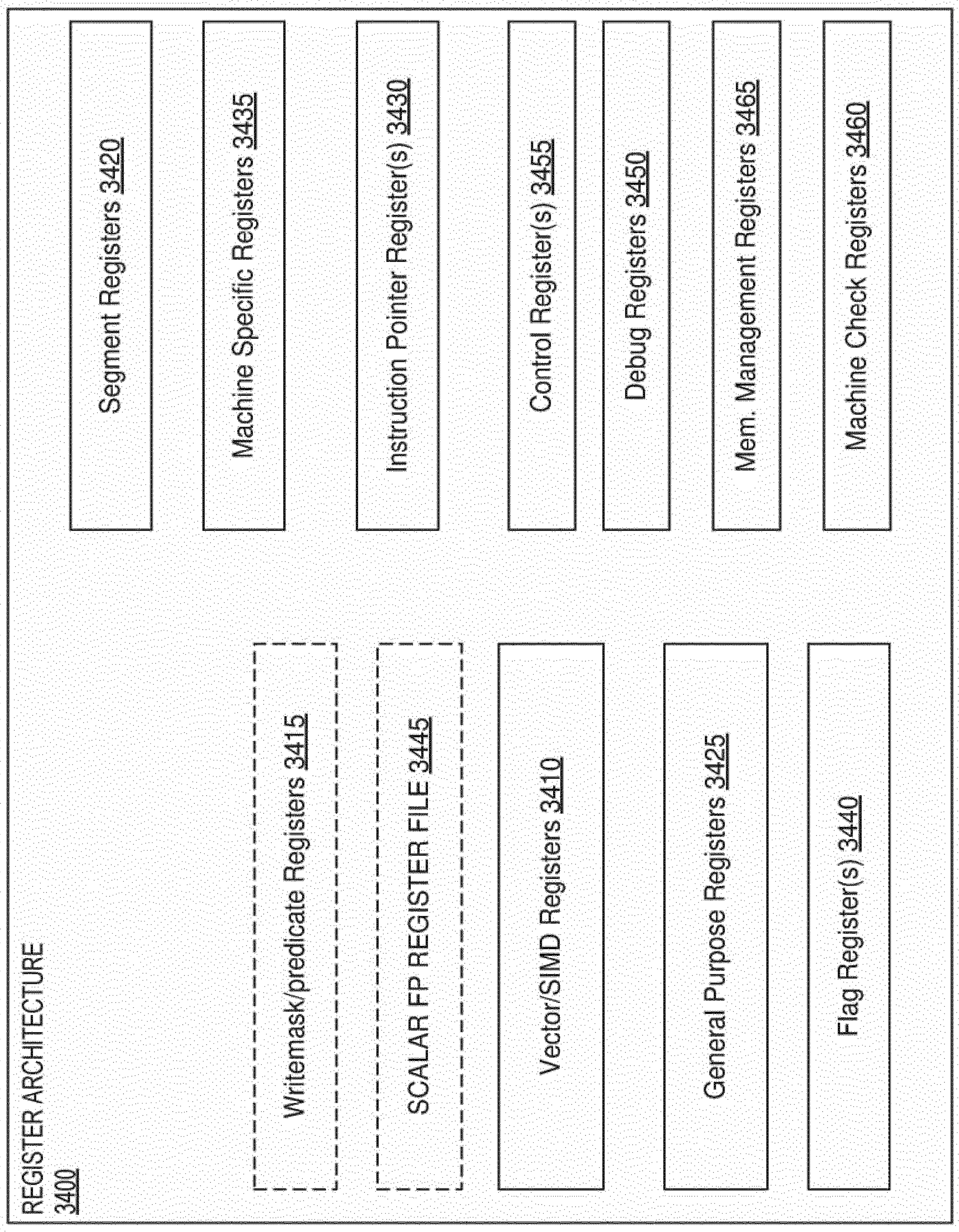

REGISTER ARCHITECTURE 3400
Writemask/predicate Registers 3415
SCALAR FP REGISTER FILE 3445
Vector/SIMD Registers 3410
General Purpose Registers 3425
Flag Register(s) 3440
Segment Registers 3420
Machine Specific Registers 3435
Instruction Pointer Register(s) 3430
Control Register(s) 3455
Debug Registers 3450
Mem. Management Registers 3465
Machine Check Registers 3460

FIG. 34

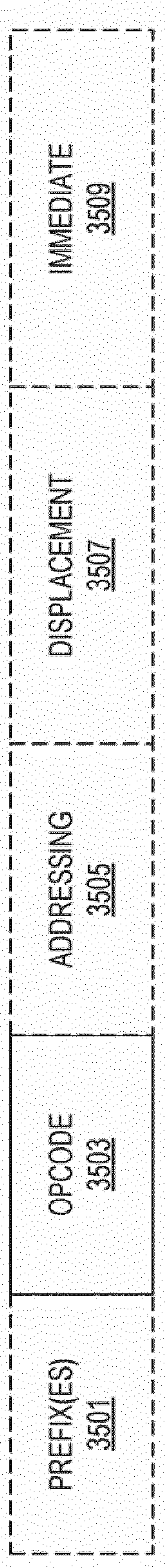

PREFIX(ES)
3501
OPCODE
3503
ADDRESSING
3505
DISPLACEMENT
3507
IMMEDIATE
3509

FIG. 35

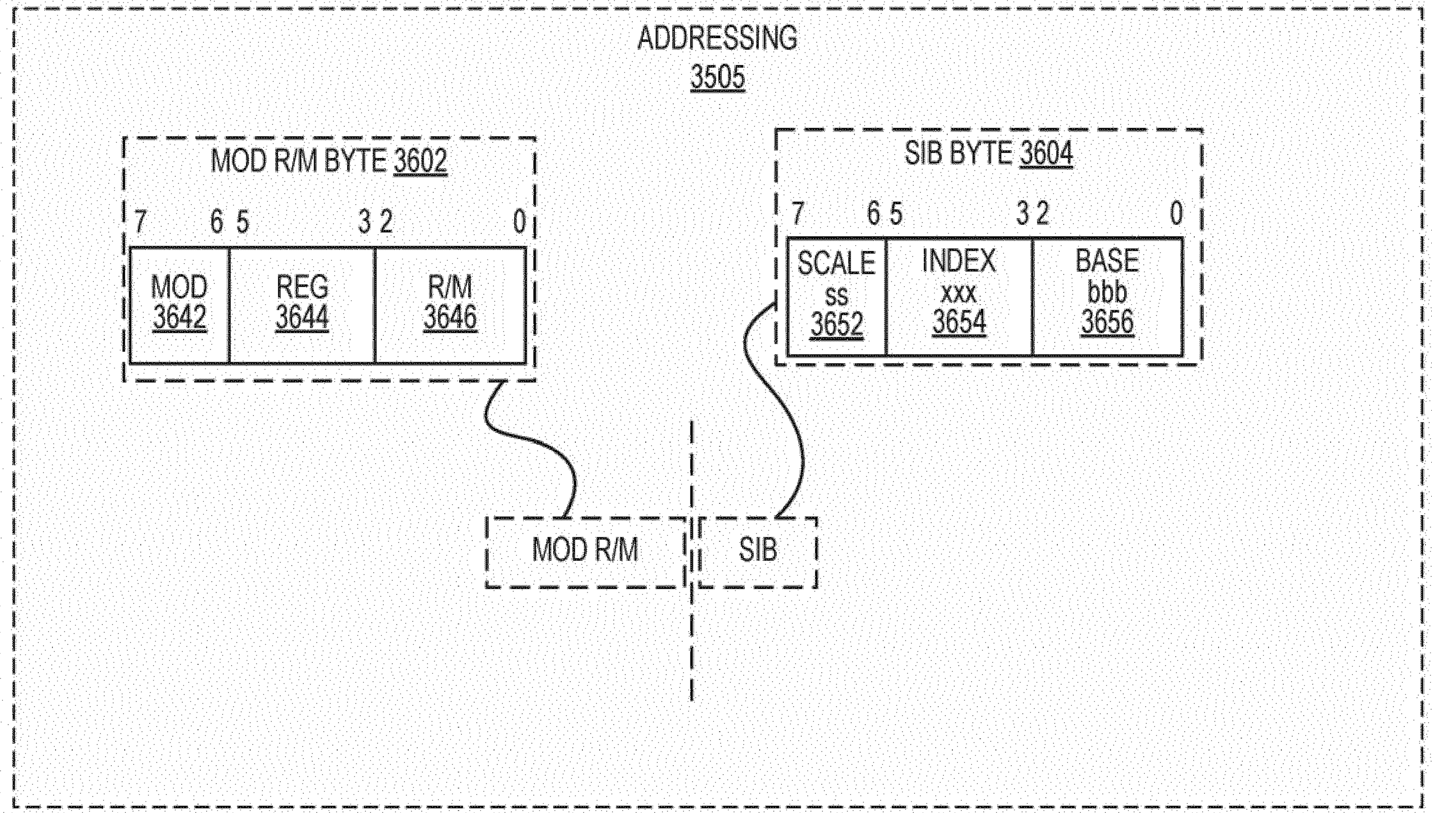

ADDRESSING 3505
MOD R/M BYTE 3602
7
6 5
3 2
0
MOD 3642
REG 3644
R/M 3646
SIB BYTE 3604
7
6 5
3 2
0
SCALE ss 3652
INDEX xxx 3654
BASE bbb 3656
MOD R/M
SIB

FIG. 36

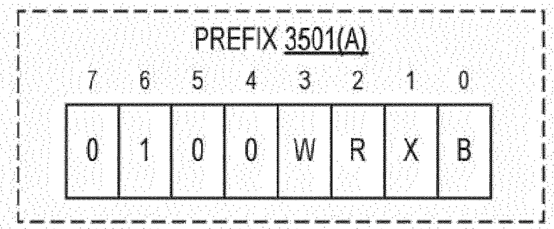


FIG. 37

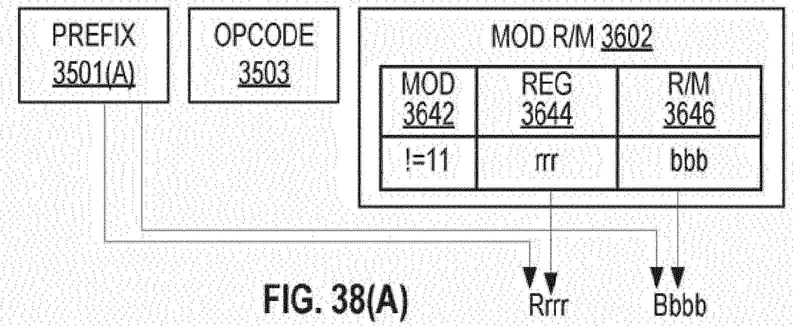


FIG. 38(A)

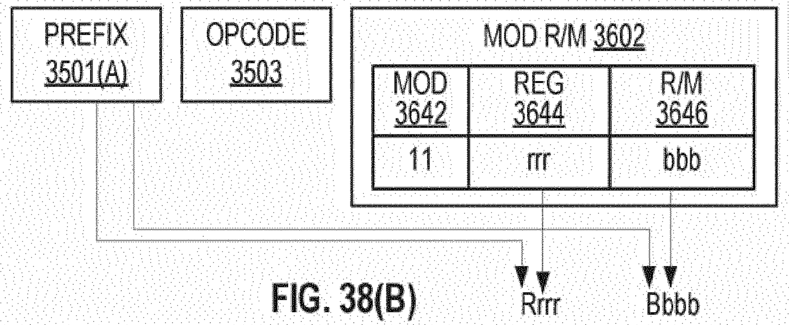


FIG. 38(B)

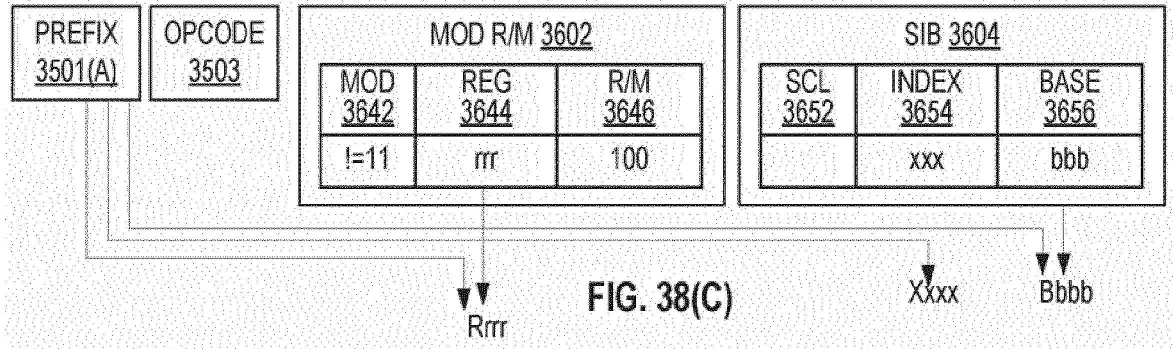


FIG. 38(C)

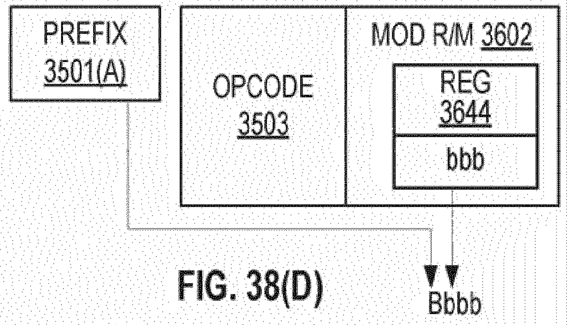


FIG. 38(D)

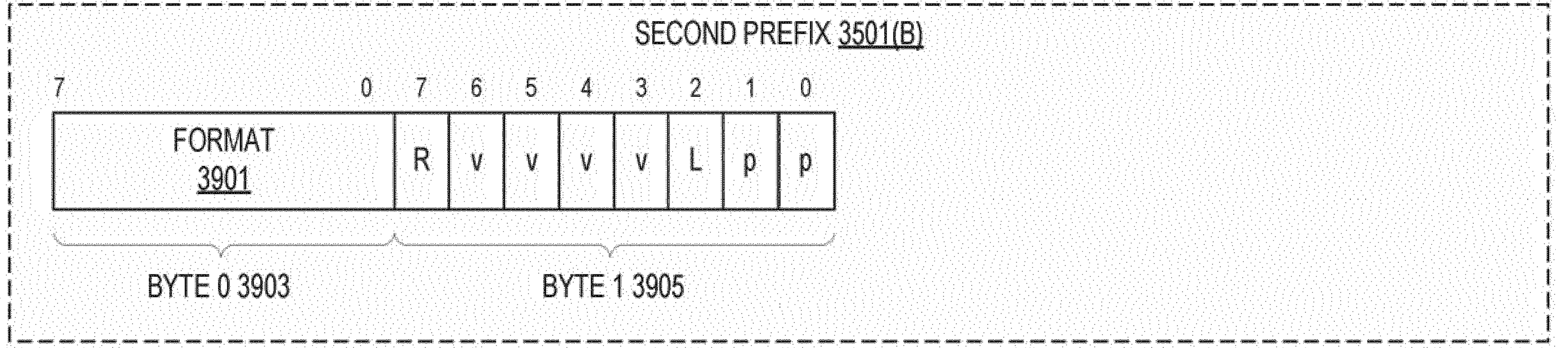


FIG. 39(A)

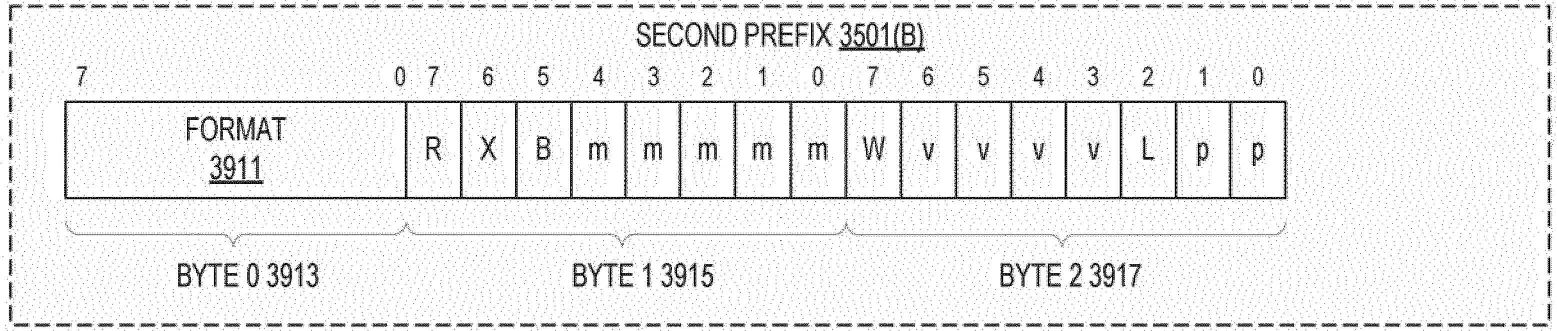


FIG. 39(B)

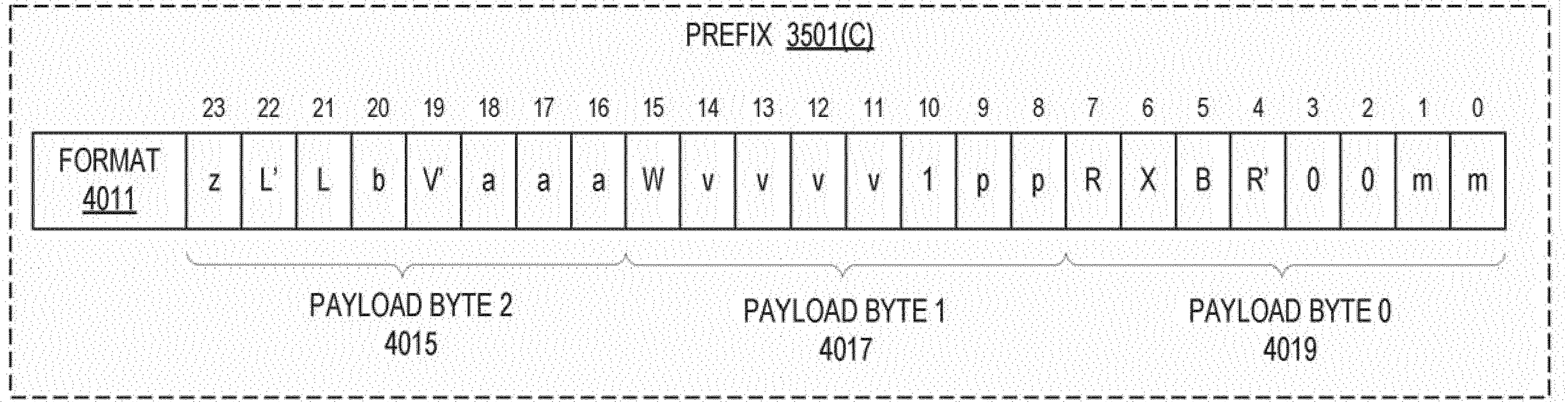

PREFIX 3501(C)
23
22
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
FORMAT
4011
z
L'
L
b
V'
a
a
a
W
v
v
v
v
1
p
p
R
X
B
R'
0
0
m
m
PAYLOAD BYTE 2
4015
PAYLOAD BYTE 1
4017
PAYLOAD BYTE 0
4019

FIG. 40

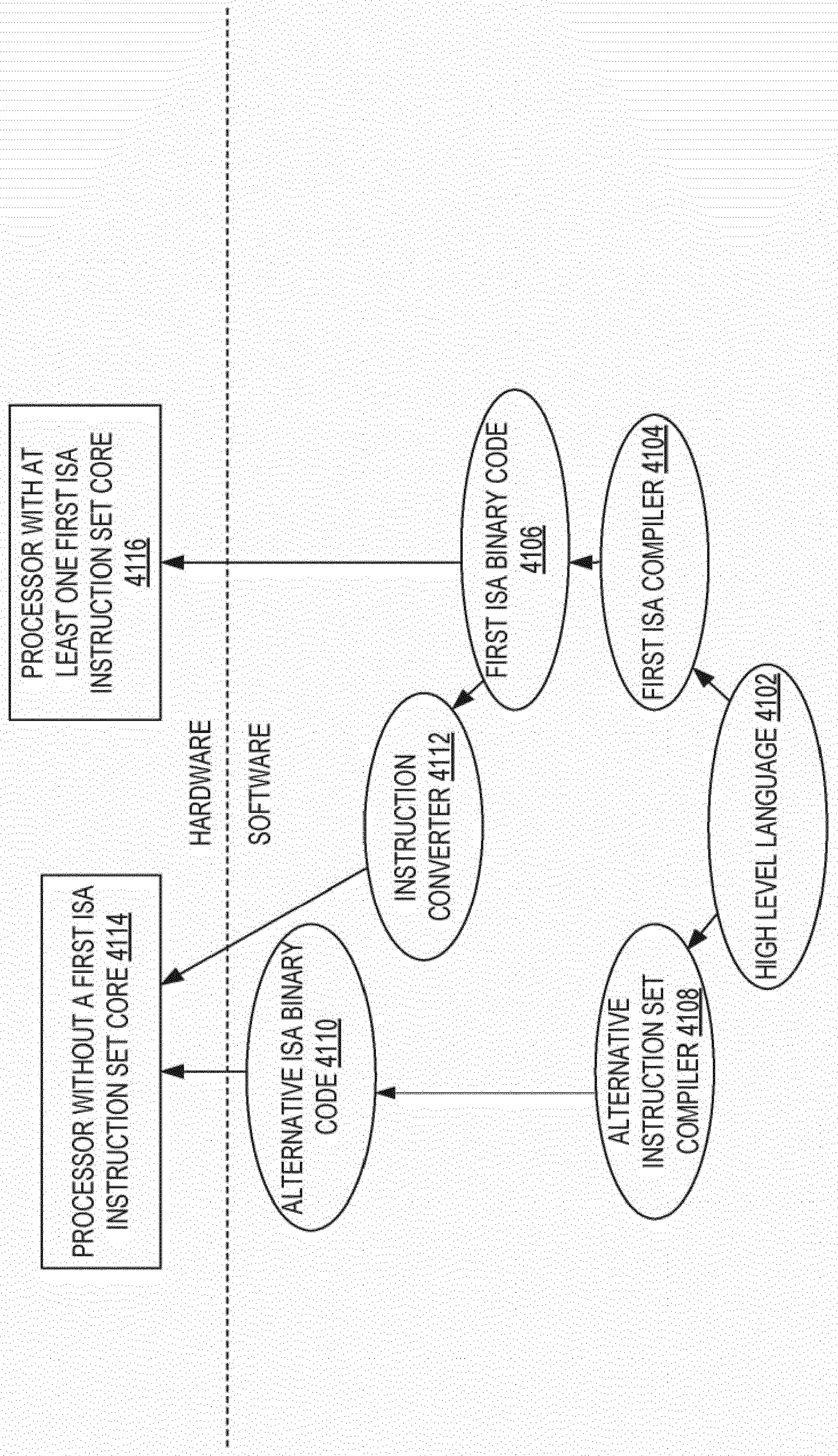

PROCESSOR WITH AT LEAST ONE FIRST ISA INSTRUCTION SET CORE 4116
PROCESSOR WITHOUT A FIRST ISA INSTRUCTION SET CORE 4114
HARDWARE
SOFTWARE
FIRST ISA BINARY CODE 4106
INSTRUCTION CONVERTER 4112
ALTERNATIVE ISA BINARY CODE 4110
FIRST ISA COMPILER 4104
HIGH LEVEL LANGUAGE 4102
ALTERNATIVE INSTRUCTION SET COMPILER 4108

FIG. 41

Europäisches Patentamt
European Patent Office
Office européen des brevets

EUROPEAN SEARCH REPORT

Application Number
EP 22 18 1066

DOCUMENTS CONSIDERED TO BE RELEVANT

Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (IPC)
X Y	US 2019/079768 A1 (HEINECKE ALEXANDER F [US] ET AL) 14 March 2019 (2019-03-14) * paragraph [0170]; figure 21 * -----	1,7 2-5, 8-11,13	INV. G06F9/30
Y	WO 2008/036944 A1 (INTEL CORP [US]; ZOHAR RONEN [US]; STORY SHANE [US]) 27 March 2008 (2008-03-27) * paragraph [0026] * -----	2-5, 8-11,13	
A	US 10 223 114 B1 (MADDURI VENKATESWARA [US] ET AL) 5 March 2019 (2019-03-05) * abstract * -----	6,11,12	

TECHNICAL FIELDS SEARCHED (IPC)

G06F

The present search report has been drawn up for all claims

1

Place of search	Date of completion of the search	Examiner
The Hague	8 November 2022	Moraiti, Marina

CATEGORY OF CITED DOCUMENTS

X : particularly relevant if taken alone
Y : particularly relevant if combined with another document of the same category
A : technological background
O : non-written disclosure
P : intermediate document

T : theory or principle underlying the invention
E : earlier patent document, but published on, or after the filing date
D : document cited in the application
L : document cited for other reasons
....
& : member of the same patent family, corresponding document

ANNEX TO THE EUROPEAN SEARCH REPORT ON EUROPEAN PATENT APPLICATION NO. EP 22 18 1066

This annex lists the patent family members relating to the patent documents cited in the above-mentioned European search report.
The members are as contained in the European Patent Office EDP file on
The European Patent Office is in no way liable for these particulars which are merely given for the purpose of information.

08-11-2022

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2019079768 A1	14-03-2019	CN 111177647 A	19-05-2020
		CN 114356417 A	15-04-2022
		EP 3651017 A2	13-05-2020
		EP 4002105 A1	25-05-2022
		US 2019079768 A1	14-03-2019
		US 2021286620 A1	16-09-2021
WO 2008036944 A1	27-03-2008	CN 101149674 A	26-03-2008
		CN 101882064 A	10-11-2010
		CN 103593165 A	19-02-2014
		CN 105573715 A	11-05-2016
		CN 109871235 A	11-06-2019
		CN 110471643 A	19-11-2019
		DE 112007001989 T5	18-06-2009
		JP 4990977 B2	01-08-2012
		JP 5475746 B2	16-04-2014
		JP 5851536 B2	03-02-2016
		JP 6162203 B2	12-07-2017
		JP 6333439 B2	30-05-2018
		JP 6487097 B2	20-03-2019
		JP 2010503935 A	04-02-2010
		JP 2012043479 A	01-03-2012
		JP 2014123390 A	03-07-2014
		JP 2016058098 A	21-04-2016
		JP 2017157225 A	07-09-2017
		JP 2018139134 A	06-09-2018
		KR 20080027454 A	27-03-2008
		RU 2009115188 A	27-10-2010
		RU 2011102731 A	20-03-2012
		US 2008077779 A1	27-03-2008
		US 2013191433 A1	25-07-2013
		US 2016139918 A1	19-05-2016
		US 2017220347 A1	03-08-2017
		US 2017220348 A1	03-08-2017
		US 2017220349 A1	03-08-2017
		US 2017322802 A1	09-11-2017
		US 2017322803 A1	09-11-2017
		US 2017322804 A1	09-11-2017
		US 2017322805 A1	09-11-2017
		US 2021216314 A1	15-07-2021
		WO 2008036944 A1	27-03-2008
US 10223114 B1	05-03-2019	CN 109582355 A	05-04-2019
		DE 102018006757 A1	04-04-2019
		US 10223114 B1	05-03-2019
		US 2019196818 A1	27-06-2019

For more details about this annex : see Official Journal of the European Patent Office, No. 12/82

ANNEX TO THE EUROPEAN SEARCH REPORT ON EUROPEAN PATENT APPLICATION NO.

EP 22 18 1066

08-11-2022

Patent document cited in search report	Publication date	Patent family member(s)	Publication date